AFRL-RI-RS-TR-2012-188

TECHNIQUES AND TOOLS FOR TRUSTWORTHY COMPOSITION OF PRE-DESIGNED EMBEDDED SOFTWARE COMPONENTS

VIRGINIA POLYTECHNIC INSTITUTE AND STATE UNIVERSITY

JULY 2012

FINAL TECHNICAL REPORT

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE

■ **AIR FORCE MATERIEL COMMAND** ■ **UNITED STATES AIR FORCE** ■ **ROME, NY 13441**

NOTICE AND SIGNATURE PAGE

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)* JUL 2012	**2. REPORT TYPE** FINAL TECHNICAL REPORT	**3. DATES COVERED** *(From - To)* JUL 2010 – DEC 2011

4. TITLE AND SUBTITLE **TECHNIQUES AND TOOLS FOR TRUSTWORTHY COMPOSITION OF PRE-DESIGNED EMBEDDED SOFTWARE COMPONENTS**	**5a. CONTRACT NUMBER** FA8750-10-1-0206
	5b. GRANT NUMBER N/A
	5c. PROGRAM ELEMENT NUMBER 63781D
6. AUTHOR(S) Sandeep K Shukla, Julien R Ouy, Mahesh Nanjundappa, Preeti Kumar, Matthew Anderson, Godwin Selvam, Matthew Kracht, Jasdeep S Malhotra, Neil Murray, Erik Rosenthal, Andrew Matsusiewicz	**5d. PROJECT NUMBER** OCXC
	5e. TASK NUMBER 11
	5f. WORK UNIT NUMBER 11
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Virginia Polytechnic Institute & State University, Virginia Tech, 1880 Pratt Drive STE 2006, Blacksburg VA 24060-6750	**8. PERFORMING ORGANIZATION REPORT NUMBER** N/A
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Air Force Research Laboratory/RITA, 525 Brooks Road, Rome NY 13441-4505	**10. SPONSOR/MONITOR'S ACRONYM(S)** N/A
	11. SPONSORING/MONITORING AGENCY REPORT NUMBER AFRL-RI-RS-TR-2012-188

12. DISTRIBUTION AVAILABILITY STATEMENT
Approved for Public Release; Distribution Unlimited. PA# 88ABW-2012-4032
Date Cleared: 19 Jul 12

13. SUPPLEMENTARY NOTES

14. ABSTRACT
This final report contains the findings from the trustworthy composition of pre-designed software components project. Since DoD software is often developed at multiple vendor sites, individual components may be verified and delivered to an integrator -- but the integration might fail because of behavioral incompatibility of the component interfaces. This work proposed behavioral types of interfaces by extracting the polychronous model from source code, and then using the theory of polychronous composition to check if the composition satisfied all of the requirements. The computation of composition properties (type inference) required novel development of a specialized Boolean theory for Prime Implicates, which provided a huge computational improvement through highly efficient generation of Prime Implicates.

15. SUBJECT TERMS
Software Engineering, Software Producibility, Component-based software design, behavioral types, behavioral type inference, Polychronous model of computation, Prime Implicates, Boolean Abstraction

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON**
					STEVEN L. DRAGER
a. REPORT	**b. ABSTRACT**	**c. THIS PAGE**			**19b. TELEPONE NUMBER (*Include area code*)**
U	U	U	SAR	93	**N/A**

Table of Contents

List of Figures

Foreword

About 10 years ago, with Prof. Rajesh Gupta at the University of California at Irvine, we embarked on a project for hardware model composition at the highest level of abstraction. At that time, due to the increased complexity of hardware designs and increased size due to the continuation of Moore's law, it started becoming extremely hard to design hardware chips in the old ways using register transfer level modeling directly. Thus transaction level modeling at various levels of accuracy came into existence, and plenty of new models were being made available in a new C++ based hardware description language called SystemC. In our project code named BALBOA, we created a component composition framework such that existing components can reflect their behavioral types at their interfaces through an introspection mechanism. Therefore, an integrator of components can choose and combine components to achieve functionality but on the way get assurance that the composition will not deadlock or will fail to deliver output signals or receive input signals at the right instant, and thereby make computational mistakes. After the limited success of BALBOA we embarked on another project at Virginia Tech for component composition but this time with meta-modeling rather than reflection. Using meta-data about components (including temporal properties as seen at the interfaces) we could also provide assurance for composition.

This led us to believe that similar techniques will apply to software composition. However, software components are not necessarily active components like hardware components. In a way, all hardware components have their own thread of computation. As clocks tick, they sample their inputs, compute, store, and produce output, and keep doing this ad infinitum. Since most hardware is synchronous, the problem is easier because the click provides a synchronization barrier. For globally asynchronous locally synchronous designs, the complication starts, where we cannot synchronize the actions of the different components with the help of a single barrier signal (clock). In software it is even more complex. First, not all software components are active objects. Some only respond to method calls, but do not actively compute. These passive objects or components execute their actions in the caller's thread, and hence are sequentially composed in the thread of the callers. So the problem there is to determine that if a number of different callers are executing the code in that component, then they do not fail in synchronization at the right moments.

However, our interest was more in active components, such that each component has its own thread of computation, and it only reacts to inputs and acts on other components by producing outputs. In the absence of a clock to synchronize the moments when inputs should be read or outputs should be produced, the inputs may not all be read at the same time, and not all outputs may be produced at the same time. Even during a single iteration of its loop, it may have conditionals that guard when an input is read, and when an output is produced. This gives rise to the notion of *Polychrony*: different inputs and outputs act at different iterations, and thus, if an interacting component cannot determine when to expect input from another one, and always expects input at times when the sender is not supposed to produce it, it may have problems of deadlock, live-lock etc.

If the loop iterations could be synchronized with barrier synchronization, this can be simplified to some extent but at the cost of performance. Hence one should try to minimize the synchronization requirements so that synchronizations are only used to prevent data races, and non-determinism. This prompted us to propose that we can use the notion of Polychrony and its notion of composition to check for these.

The only known Polychrony implementation was from the ESPRESSO project in France. We began our own implementation in the form of MRICDF, rather than SIGNAL, in order to have more control over the code of tools built, and the methodology. However, we have used the terms SIGNAL and MRICDF often interchangeably in this document.

Preface

We have heard anecdotes about the cost overrun of the F-22 due to the failure of integration of software components developed and verified by distinct vendors at seven congressional districts. We never found any documentary evidence to the truthfulness of this anecdote, but it is quite conceivable to any software engineer who has worked on a project that has more than a million lines of source code. The decomposition of requirements at the outset of the project and the agreement on the interfaces often precede the contracting out of various components to multiple vendors. However, the interfaces that are agreed upon by all the parties usually are static interfaces. This means that they agree on data types of arguments and results of methods, and possibly on sizes of variable width data type etc. The interaction diagram at the UML level capture some of the interactions between components, but more often these are not exhaustive, and they do not capture all the dynamic behaviors of the components which will be eventually integrated. This leads to integration failure, and requires redesign of components or at least their interfaces, leading to cost overruns, and time-to-market delays.

The Software Intensive Systems Producibility initiative of the Office of Secretary of Defense (OSD) looks for ways to enhance productivity of software designers of safety-critical systems. A lot of the work in this area focuses on verification and validation (V&V) because 70% of the resources of sizable software projects are spent on V&V efforts. Therefore, our project tries to provide a way of carrying out cost-effective V&V for the integration stage. One way to verify that the integration works would be to first integrate and subject the integrated system to full V & V effort. But due to the scalability problems of verification tools, this becomes quickly infeasible.

So the effort should be on verifying that the integration works without having to re-verify the components, and hence only the interactions. The concept of behavioral types comes out of this idea. One could provide behavioral types of the interfaces in many ways (e.g. state machine models, temporal logics, etc.), but the issue might be that these all require over-synchronized integration for these models to compose. We therefore chose the Polychronous model of interfaces, and leverage theories that already are known to work in practice. However, we found two problems: (i) besides the single source risks, the only Polychrony implementation was not indigenous and therefore not always necessarily useful to DoD, and (ii) the theory of Polychrony was never tested for solving this problem as it was invented for model driven code synthesis, and hence there were gaps in the theory – especially in scaling issues. So we had to work on developing our own Polychrony implementation, which we already started one summer at Air Force Research Laboratory (AFRL) when the PI was a summer faculty visitor at AFRL with Mr. Steve Drager. However, we needed to develop this theory further to accommodate the needs of scalable composition checking, as described in the results section.

Overall, this project was a very good learning ground for the advanced model driven software engineering theory that European projects have worked on for over two decades and the United States academia did not pay a lot of attention to until very recently. The theories and models of the French synchronous language community led to tools such as SCADE, which is now a certified tool used in avionics software synthesis, albeit only a small percentage of the avionics software at Airbus. It is a triumph of model driven software design, synthesis and verification research.

We therefore not only brought that technology into the DoD knowledge space, we actually improved the theory, and we have provided alternative synthesis algorithms. We think that is the most important contribution of this project.

The practice of compositional design with behavioral interface types, examined at the end of this project, has not moved past prototyping because we need to partner with a DoD software vendor to deliver an industry strength software. That is one of the most important failures of this project.

Overall, we think we can build on these results in partnership with a DoD software vendor to implement the theory and knowledge we have gained in this project.

Acknowledgement

We acknowledge the support of Mr. Steven Drager from the Air Force Research Laboratory throughout the project. Besides funding this project, his group has continuously provided feedback and helped us with many different technical and nontechnical issues related to this project. We also acknowledge the support of Dr. Michael May from the Office of the Secretary of Defense, Office of the Assistant Secretary of Defense for Research and Engineering (OASD(R&E)) who has also provided very useful feedback during the semi-annual review meetings and his site visit to Virginia Tech. Mr. William McKeever from AFRL also provided various feedback and we are grateful for all the generous help and directions provided by all of them.

The ESPRESSO research team at the French National Institute of Computer Science and Automation (INRIA), particularly Jean-Pierre Talpin, has provided valuable advice and feedback on the use of Polychrony and polychronous model of computation. Jens Brandt from the Quartz project at the Technical University of Kaiserslautern, Germany also helped in many ways, as the theme of this project is also of great interest in Europe.

1 Summary

This section overviews the main topics covered in this project. In order to keep it brief the major accomplishments of this project, and some disappointments of the project, are listed but not described here. Further details will be provided in the subsequent sections.

1.1 Major Highlights

- New Methodology for Compatibility Checking between separately developed components
- Prime Implicate-based algorithm for behavioral compatibility checking
- Improvement of the Embedded Code Synthesis (EmCodeSyn) tool to extract behavioral types, transform Multi-Rate Instantaneous Channel Connected Data-Flow Actor Model (MRICDF) code to Signal
- C-to-SIGNAL extraction tool implementation to extract behavioral types from C-based components
- Two Case studies (ArduPilot, AutoSar)
- New Improved algorithms for Prime Implicate Generation
- Faster Prime Implicate Generation by Filtering and Decomposition
- Graph based Heuristic for Epoch Calculus with incremental algorithm

1.2 Disappointments

- The project lasted only 18 months. It takes about a year to hire and train a postdoctoral fellow. By the time he was becoming productive, the contract was over. More advances could have been made if the postdoctoral personnel were already trained in the specific area of polychronous modeling and synthesis.
- The project ended before it was at a stage where it could be transferred to a DoD software vendor for creating a robust tool. Transition to a DoD vendor is in the works.
- Expressing interface behaviors that depend on numeric ratios (e.g., one input comes every three times another input comes) is harder to capture in the standard polychronous model (it is possible by modeling a counter but the model gets large). The intent was to extend this developed theory to polychronous models' affine clocks to make more efficient behavioral types, but the time frame did not allow this.
- The task to verify if regular expression subsumption could be used for simple interfaces was not pursued long enough to achieve results.
- The case studies of ArduPilot and Autosar platforms are not yet 100% complete. Only selected parts of the platforms were modeled in Polychrony (albeit nontrivial percentage) and while experimentation with composition and error injection was achieved the goal of 100 % coverage was not achieved.

2 Introduction

2.1 Motivation

Building large mission critical software systems often requires use of software components designed by multiple vendors or by distinct groups within a single organization. This allows for *concurrent engineering* and *reuse* of existing components, thereby reducing system development time, and cost. Combining independently developed software components to create trustworthy software systems, even when the components are pre-verified for correct functionality, is a formidable challenge. This is especially true of *safety-critical* applications, which require a much higher level of confidence in their correctness and reliability than other kinds of applications. A common practice in the United States is to use tools such as MATLAB/Simulink or Labview to create simulation models followed by either hand translation or code generation from these models. However, these modeling languages are not endowed with proper formal semantics, and the generated code needs thorough (and expensive!) verification before certification. Formal methods provide a framework for transforming this practice to one of generation of *provably correct* software. However, even when the individual components are synthesized with such a *correct-by-construction* methodology, the various information items obtained during such synthesis/compilation are not usually exported with the component, resulting in loss of information (such as *temporal behavior at the interface of the components*) available in the original formal specification of the components. Larger software systems are constructed by composing these components from a library. Due to the loss of information one often requires reverse engineering of the components to reconstruct that crucial information, some of which cannot always be recovered. Even though the static data-types of the variables and the methods at the component interfaces (in strongly typed languages) provide type correctness of a composition, for concurrent reactive software, mismatches in temporal behaviors of an output variable at the interface of one component, and that of an input variable of another component, either demands to be bridged by intervening protocols, or they must not be composed. In order to extend a correct-by-construction technique for component synthesis to the problem of safe composition, one needs to also export relevant information created during the synthesis process as *behavioral types* of the components. This project aimed at developing the structure of such behavioral types, and methods for safely composing pre-synthesized components endowed with behavioral types. Novel techniques and algorithms based on computing *Prime Implicates* for propositional *Boolean theories* and Boolean Theories with *Presburger arithmetic* were also developed to address the challenges of component composition leading to a correct-by-construction composition technique.

2.1.1.1 Impact and Transformative Potential

This project brought together experts from formal specification and specification-driven synthesis, and from automated deduction and prime implicate computation, to solve an important problem of trustworthy large scale mission critical software development. Interaction with the U.S. Air Force Research Laboratory, Boeing, and Lockheed Martin indicates that embedded software is mostly programmed manually. Even when synthesized, the code is not provably correct, and expensive verification is required. Appropriate formalism is rarely used, and cost-overruns and delays in final release, for example, in the production of the F-22, are common. This research has the potential to significantly improve the reliability of their software development processes while substantially reducing costs.

2.2 Structuring of the Reported Results

This report details the work accomplished in the project and describes the work done in moderate detail. For the convenience of the reader, the sections of this report are numbered based on the section numbers in the original proposal where the tasks were described at the inception of the project.

2.3 Statement of Work from the Original Proposal

- 4.1. Development of Behavioral Type for Software Components
  - 4.1.1. Develop a theory of type for temporal behavior at the interface of software Components
  - 4.1.2. Develop methods for behavioral type extraction from MRICDF specification of a software component
  - 4.1.3. Develop methods for behavioral type extraction from C code
- 4.2. Development of Behavioral Type Inference Algorithm as a proof technique for Trustworthy composition
  - 4.2.1. Reduction of behavioral type inference to Prime Implicate extraction problem
    - 4.2.1.1. Investigate if SAT Modulo Theory (SMT) techniques will refine behavioral Types
- 4.3. Implementation of Behavioral type extraction algorithm in EmCodeSyn Tool
  - 4.3.1. Implementation of the type extraction from MRICDF models into EmCodeSyn Tool
  - 4.3.2. Possible attempt to automate the extraction of type from C code
- 4.4. Implementation of Behavioral type inference algorithm
  - 4.4.1. Use Prime Implicate extraction tools for type inference implementation within EmCodeSyn
- 4.5. Development of Prime Implicate extraction algorithms
  - 4.5.1. Tuning current Murray-Rosenthal Algorithm to only produce certain specific kind of prime implicates rather than all, thus reducing time required (In particular unitary positive prime implicates)
  - 4.5.2. Developing an incremental pi-trie algorithm for prime implicates such that every subsequent iteration of the prime implicate extraction will not require building a completely new pi-trie from scratch, thus reducing time requirement
  - 4.5.3. Development of interactive prime implicate computing algorithm (user provides hints from the structure of the software specification as to what might be possible prime implicate which may be easy for user to guess from the MRICDF)
- 4.6. Case Study
  - 4.6.1. Example from Virginia Tech Unmanned Vehicle Control team will be considered as demonstration software on which the composition technique will be on going

3 Methods, Assumptions and Procedures

This section provides some background information on the methods and techniques the research is based on.

3.1 Programming Model, and Synthesis Technique

In the recent past, the group has developed a programming model called Multi-Rate Instantaneous Channel Connected Dataflow Actor Model [1, 2, 3, 4], to capture the specification of a reactive embedded software. A visual specification and component code synthesis tool called EmCodeSyn [2] was also developed which accepts MRICDF specifications in visual or textual form, and produces C-code that is correct-by-construction with respect to its MRICDF specification. Similar to Esterel [18], Lustre [19], and SIGNAL [6], the model of computation of MRICDF is based on the synchrony hypothesis [20], which provides a suitable abstraction from computation and communication time, and allows one to focus on the dataflow and computation functionality of the required software. Almost all of these formalisms with the exception of MRICDF are developed in Europe. Airbus [24], Renault, and other European avionics and automotive companies claim to generate a large percentage of their control software using these formal approaches. Even though they have been successful in developing modules through these methodologies, for composition of modules they use an over simplified composition model based on a Time-Triggered Architecture [17]. This requires that each component itself be time triggered leading to a number of optimality problems as pointed out in [1, 4]. The semantics of MRICDF is not time triggered but rather event triggered, leading to more optimal code synthesis [1]. Time triggered composition has another problem other than optimality. It requires precise clock synchronization which results in a large overhead. It would be convenient to achieve both optimal implementation of individual components and avoidance of the overhead of time synchronization over a distributed platform. Therefore, constructing large software systems from components synthesized with the proposed tool is much more challenging, but the benefits outweigh these difficulties as will be addressed in this project.

The programming model of MRICDF is that of a collection of concurrent processes described by data flow relations on infinite streams of data values.[1] The synchronization requirements between these streams are expressed either implicitly by the data flow relations or by explicit constraints. When sequential embedded software is to be synthesized, both data flow relations — computation — and synchronization constraints — control — must be considered. This is the crux of the compilation/synthesis process for MRICDF. This programming model is more suitable for reactive systems compared to other specification models such as temporal logics, composition of automata (such as I/O Automata) etc., because it abstracts away timing issues but most importantly, it makes specification of synchronization between concurrent activities within each component much easier than those other methods. The expression of synchronization between concurrently acting behaviors within a system is a major source of errors (deadlocks, live-locks, violation of mutual exclusion etc.) in other formalisms.

[1]Most embedded applications work on infinite streams of data.

Given a specification (visual or textual) in MRICDF, a compilation algorithm must decide whether there exists a deterministic sequential/multi-threaded code satisfying the constraints, and if so, whether it is unique. If not — and thus nondeterministic — the user must provide additional constraints to make it so. If this effort fails, the specification is rejected by the compiler. In the process of determining implementability and subsequent synthesis, the compiler creates a *Boolean theory* and computes its prime implicates.

3.2 Boolean Theory and Prime Implicates

A Boolean theory is a set of Boolean clauses. Let the theory B be defined over a set of Boolean variables X. Then $[X \rightarrow \{0,1\}]$ denotes the space of all assignments to variables in X. An assignment, $f \in [X \rightarrow \{0,1\}]$, is a model for theory B, if and only if by assigning the Boolean values to all variables $x \in X$ as $f(x)$, one can satisfy all clauses in B.

A Boolean theory that is satisfiable has at least one such model. A prime implicate of a Boolean theory is a disjunctive Boolean clause C such that any model of B also satisfies C and there is no C_1 such that $C_1 \rightarrow C$ and any model of B also satisfies C_1.

Given an arbitrary Boolean theory, computing the prime implicate is often of exponential complexity. Most current algorithms also require that the Boolean clauses in the theory be first converted into a conjunctive normal form (CNF) before applying the algorithm. The recent work of Murray and Rosenthal [34] has derived a new algorithm that can produce prime implicates (and an implicit representation of all prime implicates) of a Boolean theory where the clauses can be in any arbitrary form. However, this algorithm is time consuming.

It is, however, expected that this algorithm may be sped up substantially as the current algorithm is *agnostic* of any special characteristics of the Boolean theory that are generated from MRICDF models during computations of their master triggers.

It has been shown in the past that algorithms that are agnostic of the special nature of the inputs on which the algorithm is applied have higher time and space complexity than algorithms that take into account the special nature of their inputs. For example, finding the chromatic number of a graph is known to be NP-Complete, but if one knows that the only graphs needed to compute the chromatic number belong to a special class of graphs called "Perfect Graphs," then one can come up with special algorithms which can compute the chromatic numbers in polynomial time. Similarly, the famous SAT problem that is well known to be NP-Complete can be shown to be solvable in polynomial time, if the clauses obtained belong to the class of HORN clauses. Also, there is a notion of localization of problem instances. For example, if the variables that occur in multiple clauses can be limited to reappear in no more than k clauses. One may say that that SAT problem is k-bounded. In such a case, one can devise faster algorithms for solving SAT.

Since the Boolean theories that are generated from MRICDF are very localized, in the sense that a clause $x \leftrightarrow y \vee z$, appears only when y and z are inputs to a merge actor in an MRICDF model, one can find such locality properties. As a result, Murray and Rosenthal's algorithms to compute prime implicates may not exploit such locality (referred to as ’regularity’) and it may be required to devise new algorithms that work much faster computing prime implicates for such instances.

Members of the project team have past expertise in solving problems with such locality properties for various graph and satisfiability problems. So as the research progresses, it is highly likely that the discovery and characterization of the Boolean theories that appear during analysis of MRICDF models can be solved with much faster algorithms.

Prime implicates enable construction of a hierarchical control structure that creates a deterministic schedule of all the computations which are consistent with the control constraints. If non-Boolean constraints — for example, $x > 10$ — are replaced by unrestricted Booleans, the resulting theory becomes a conservative abstraction of the more elaborate theory with further expressiveness. The latter case would provide better leverage in optimizing the control structure and in reducing redundant paths. To this end, the combination of prime implicates algorithms and SAT Modulo Theory (SMT) solvers [25] is being investigated.

4 Results and Discussion

4.1 Development of Behavioral Type for a Software component

4.1.1 Development of a theory of type for temporal behavior at the interface of a software Component

Overview: It was proposed, as a behavioral type, to consider the polychronous model of a component. From this model, one could extract and manipulate significant information like clock relations and data dependencies and check for the compatibility of the components. The compatibility of polychronous components – their ability to be composed – depends on the absence of dependency cycles and the calculability of common clocks and signals. The framework and the compatibility property have been designed with the goal of composition, thus the computations of the properties benefit from modular decomposition. The theories that are involved in this work will be described as this report progresses.

4.1.1.1 Preliminary Definitions

During the execution of programs, computational activities evolve over time, but not necessarily at the same pace in every concurrent thread of execution. It is required to track the computation's progress as a partially ordered set of logical instants. It is partially ordered because it may be necessary to keep some events unordered – e.g., two concurrent actions should be able to run independently. The computation activities happening to each signal/variable in the system are termed events.

Definition 1 (Event) *Ξ denotes the set of all events. $\leq$ is a preorder on Ξ: $e \leq f$ means that e occurs before or concurrently with f. $\sim$ is the equivalence relation based on $\leq$: $e \sim f$ means that e and f occur simultaneously hence termed synchronous events.*

Definition 2 (Logical Instant or Instant) *Y denotes the quotient of Ξ by $\sim$ as the set of instants. Thus a logical instant is a maximal set of events that are synchronous.*

Note that synchronous events may have data dependencies and hence may have a partial order inside the instant, different from $\leq$. This different order serves the generation of sequential code.

Definition 3 (Partial order on Instants) *$Y/\leq$ is the partial order on instants based on $\Xi/\leq$:*

$$\forall a, b \in Y, a \leq b \Leftrightarrow \exists \alpha \in a, \beta \in b \mid \alpha \leq \beta$$

The above definition lifts the $\leq$ and the corresponding strict order $<$ to the set of logical instants to compare the order of happening of some of the logical instants.

The entities of the model are signals, defined on instants. A signal is a succession of values, happening at some specific instants. This specific succession of instants is the clock of the signal.

Definition 4 (Signal) *Let T be a set of values – the type and* $T_\perp = T \cup \{\perp\}$ *its extension with a special value meant to represent absence, a* signal *is a function* $Y \to T_\perp$. *For all instants in* $\gamma \subset Y$ *such that* γ *is a total order in Y, it associates a value from T, and for each instant of* $(Y-\gamma)$ *it associates the absence value:* $\perp$. *A maximal such* γ *is the* domain of definition *for the signal.*

$$x : \Upsilon \to T_{\perp}$$
$$x : \gamma \to T$$
$$x : \Upsilon - \gamma \to \{\perp\}$$

A signal x is defined over a total ordered set of instants:

$$\forall p, q \in \Upsilon,$$
$$x(p) \neq \perp \wedge\, x(q) \neq \perp \Longrightarrow p \leq q \vee q \leq p$$

Let us define a characteristic function σ *that tells if a signal x is present or absent at any given instant t in Y.*

$$\sigma\left(\Upsilon \to T_{\perp} \to \Upsilon \to \{true, false\}\right)$$
$$\sigma(x)(t) = true\ if\ x(t) \in T$$
$$= false\ if\ x(t) = \perp$$

Error! Bookmark not defined.

Given a signal x, one can also define a clock signal $\hat{x}$*, clocks and characteristic functions play the same role. Characteristic functions will be used in formal definitions and clocks as in this example in SIGNAL.*

$$\hat{:}(clock) : \left(\Upsilon \to T_{\perp} \to \Upsilon \to \{true, false\}\right)$$
$$\sigma(x)(t) = true\ if\ x(t) \in T$$
$$= false\ if\ x(t) = \perp$$

Thus $\sigma(\tilde{x}) = \sigma(x)$ *and two signals x and y are synchronous if and only if there exists no instance such that, x is present and y is not present and vice versa.*

The notation $x(t_x^0)$ *defined as such:* $x(t_x^0) \in T$ *and* $\forall t \in \Upsilon, t < t_x^0 \Rightarrow x(t) = \perp$ *will also be used. In other words, it is the value for signal x at its initial logical instant.*

Another notational convenience is nonstandard. The notation $x(t_x^{-1})$ *defined as such:* $\forall t \in \Upsilon$, *such that* $x(t) \in T$ *and* $t \neq t_x^0$, *where* $t_x^{-1} \in \Upsilon$ *such that* $x(t_x^{-1}) \in T$ *and* $\forall p \in \Upsilon, t_x^{-1} < p < t \Rightarrow x(p) = \perp$.

Accordingly t_x^{-1} is actually not an arithmetic subtraction from *t*, but denotes the last logical instant before the instant *t*, where the signal was defined. The processes are based on relations between clocks and signals. These relations or constraints define a system in which inputs are provided from the environment and outputs are computed by the system.

Definition 5 (Process) *A process is a tuple ($\mathcal{V}$, $\mathcal{N}$, $\mathcal{C}$, $\mathcal{D}$)*

- *$\mathcal{V}$ is a set of signals.*

- *$\mathcal{N}$ is a mapping associating names to signals of $\mathcal{V}$.*

  $$\mathcal{N} \subseteq \mathcal{V} \times Name$$

  Note that same signal may be named differently in different processes, hence $\mathcal{N}$ is a relation. It is used in composition of processes, or in instantiation. Two identical processes can be instantiated with different $\mathcal{N}$ and then work on different signals.

- *$\mathcal{C}$ is a system of equations that constraints σ functions (i.e., the clocks) with signals. Therefore $\mathcal{C}$ is constraints on the* control flow *of the system.*

  $$\forall e \in \mathcal{C}, \exists x \in \mathcal{V}, \exists t \in \Upsilon$$

  $$e := \sigma(x)(t) = \overset{x}{e_t}(\{x_i(t')\},\{\sigma(x_j)(t)\})$$

  where, $\overset{x}{e_t}$ is a Boolean operation on any signals $x_j \in \mathcal{V}$ or clock $\sigma(x_j)$ at the instants t and t'=t or $t_{x_i}^{-1}$ – that defines $\sigma(x)(t)$.

  Note that it can be shown that if a clock constraint refers values of signals at earlier instants, one can always rewrite C so that it depends on the immediate past instant t_x^{-1}. Thus the relations in $\mathcal{C}$ only refer to current and immediate past instants.

- *$\mathcal{D}$ is a system of equations that constrains the values taken by the signals together. Thus $\mathcal{D}$ is the* data flow *specification of the process.*

  $$\forall g \in \mathcal{D}, \exists x \in \mathcal{V}, \exists t \in \hat{x} \text{ such that}$$

  $$g := x(t) = \overset{x}{g_t}(\{x_i(t')\},\{\sigma(x_j)(t)\})$$

  where, $\overset{x}{g_t}$ is a valid function on the type of the values taken by the signals $x_i \in \mathcal{V}$ or clock $\sigma(x_j)$ and the instants t and t'=t or $t_{x_i}^{-1}$ – that defines x(t).

Some signals and clocks may remain unconstrained or undefined, because they are the free variables of the systems. The undefined clocks determine the pace of the program and the undefined signals will incrementally determine the values of all other signals. Undefined signals may be designed as inputs although some inputs may have constraints.

Example 1 *Let us study a SIGNAL process,* `biadder` *shown in Figure 1, and its representation in this formalism in Figure 2. This process consists of two concurrent threads that accumulate values from input* `x1` *on* `y1` *and from input* `x2` *on* `y2`*. On a specific instant defined as when both inputs carry the value 0, the threads synchronize and the output* `s` *carries the sum of both accumulations.*

```
process biadder = (? integer x1 , x2 ; ! integer y1 , y2 , s ;)
(| y1 := y1$ init 0 + x1
 | y2 := y2$ init 0 + x2
 | s := ( y1 when s ) + ( y2 when s )
 | s ^= ( x1 = 0) ^= ( x2 = 0)
 |)
```

Figure 1: Process biadder - a SIGNAL implementation

$$\mathcal{V} := \{x1, x2, y1, y2, s, (x1 = 0), (x2 = 0), (y1 \text{ when } s), (y2 \text{ when } s)\}$$

$$\mathcal{N} := \{(x1, "x1"), (x2, "x2"), (y1, "y1"), (y2, "y2"), (s, "s")\}$$

$$\begin{aligned}\mathcal{C} := \; & \forall t_1 \in \Upsilon, \sigma(y1)(t_1) = \sigma(x1)(t_1) \\ & \forall t_2 \in \Upsilon, \sigma(y2)(t_2) = \sigma(x2)(t_2) \\ & \forall t_3 \in \Upsilon, \sigma(s)(t_3) = \sigma(x1)(t_3) \wedge (x1 = 0)(t_3) \\ & \qquad \sigma(s)(t_3) = \sigma(x2)(t_3) \wedge (x2 = 0)(t_3)\end{aligned}$$

$$\begin{aligned}\mathcal{D} := \; & \forall t_1 \text{ such that } \sigma(y_1)(t_1) = 1, y1(t_1) = y1({t_1}_{y1}^{-1}) + x1(t_1) \\ & \forall t_2 \text{ such that } \sigma(y_2)(t_2) = 1, y2(t_2) = y2({t_2}_{y2}^{-1}) + x2(t_2) \\ & \qquad y1(t_{y1}^0) = x1(t_{y1}^0) \\ & \qquad y2(t_{y2}^0) = x2(t_{y2}^0) \\ & \forall t_3 \text{ such that } \sigma(s)(t_3) = 1, \; s(t_3) = y1(t_3) + y2(t_3)\end{aligned}$$

Figure 2: Control flow and Data flow constraints of process Bi-adder

Start with $\mathcal{V}$ and $\mathcal{N}$. Not every signal is given a name. Only those that need to be defined or that belong to the interface are given names. So the `biadder` *process manipulates some signals with no name, for example the Boolean signal (x1 = 0), this signal has a clock (which is $\hat{x}$ the clock of x) and can be used in an equation. y1\$ and y2\$ are not signals. Since signals contain the values for each instants over which they are defined, y1\$ is just a way to design $y1(t^{-1})$.*

$\mathcal{C}$ contains four equations derived from the SIGNAL code. The first two synchronize x1 with y1 and x2 with y2. The semantics of SIGNAL, would induce the equations $\forall t \in \Upsilon, \sigma(y1)(t)=\sigma(y1)(t) \wedge \sigma(x1)(t)$ for the ":=" operator and $\forall t \in \Upsilon, \sigma(y1)(t)=\sigma(x1)(t)$ for the "+" operator. This was simplified using Boolean rules.

The line 5 of the SIGNAL process would induce the equations $\forall t \in \Upsilon, \sigma(y1 when\ s)(t)=\sigma(s)(t)$, the same for y2 and $\forall t \in \Upsilon, \sigma(s)(t)=\sigma(y1 when s)(t) \wedge \sigma(y2 when s)(t)$. Since "y1 when s" and "y2 when s" are not used in the other formula, and the definition of "s" is a tautology; these equations are not considered.

The last line of the process induces two clock equations in $\mathcal{C}$. The clock of s is defined by two different equations and relies not only on clocks but also on values (the values of x1 and x2).

$\mathcal{D}$ contains the definitions of the signals, those definitions hold only when their clock is true (given by the function sigma). Again, definitions of "x = 0" and "y when s" are not given since they do not serve the comprehension of the example. Since signals are all of type integer, the + operator is valid as the addition of integers.

For a process with a system of clocks, one can define a clock hierarchy. A hierarchy is a stronger representation of clocks induced by the clock calculus [48]. If a hierarchy has a unique greatest element, this element is called 'root' and the hierarchy is a tree. It may also have several roots that form a forest. Building a tree or a forest helps determine if the individual states are endochronous or weakly-hierarchical and then to check isochrony of the system [53]. It also constructs a schedule for an implementation of the system.

Definition 6 (Clock Hierarchy) *The control flow constraints $\mathcal{C}$ of a process induces a set with a relation on clocks of signals – the clock hierarchy $\mathcal{H}= (\hat{\mathcal{V}}, \geq)$ defined by the following rules:*

1. *for all multi-valued signals $x \in \mathcal{V}$, and all values u that appear on the right hand side of the data flow or control flow constraints $\mathcal{D}, \mathcal{C}$:*
   $\hat{x}, (x = u) \in \hat{\mathcal{V}}$
   $(\hat{x} \succ (x = u)) \in \mathcal{H}$
2. *if $\mathcal{C} \models \forall t,\ \sigma(b)(t) = \sigma(c)(t)$ then $(\hat{b} \geq \hat{c}), (\hat{c} \geq \hat{b}), \in \mathcal{H}$ written $(\hat{b} \sim \hat{c})$*
3. *if $\exists g \in \{when, default\}, \mathcal{D} \models b1(t)=g(c1(t),c2(t))$ and $(\hat{b2} \geq \hat{c1}),(\hat{b2} \geq \hat{c2}) \in \mathcal{H}$ then $\hat{b2} \geq \hat{b1} \in \mathcal{H}$.*

Definition 7 (Root) *The Clock Hierarchy of a nonempty process is a partially ordered set with at least one greatest element. A signal r is a root in the hierarchy if $\forall x \in \hat{\mathcal{V}},\ x \geq r \in \mathcal{H} \Rightarrow x = r$. A signal r^o is a unique root of $\mathcal{H}$ if $\forall x \in \mathcal{H}, r^0 \geq r$.*

The system of equations $\mathcal{C}$ and $\mathcal{D}$ induces a graph on signals, the instantaneous dependency graph [45]. This graph represents the relation of dependencies between signals. This graph says that in some logical instant, a signal cannot be computed without knowing the value or presence of a second signal. It is important that such dependency graphs have no constructible cycle. If a cycle exists in the graph, at least two signals are mutually dependent and they cannot be calculated.

Definition 8 (Exclusive clocks) *In a Hierarchy, two clocks are exclusive # if they depend of two different values of a same signal.*

$$\forall x, y \in \mathcal{H},\ x \# y \Leftrightarrow \exists z \in \mathcal{H};\ u, v \in domain\ (z)\ |u \neq v \wedge [z=u] \geq x \wedge [z=v] \geq y$$

Definition 9 (Dependency graph of a process) *Systems $\mathcal{C}$ and $\mathcal{D}$ induce a graph on signals: the dependency graph $\mathcal{G} = (\mathcal{V}, \rightarrow)$ where $\rightarrow$ is the relation of dependency: $x(t) \rightarrow y(t)$ means that $\mathcal{C}$ contains an equation $\sigma(y)(t)=e_t^y(x)(t)$ or $\mathcal{D}$ contains an equation $y(t)=f_t^y(x)(t)$. Let $\rightarrow^+$ denote* *transitive closure of* $\rightarrow$*. The process is cyclic iff,* $\exists t \in Y, \exists x, y \in G, x \neq y, x(t) \rightarrow^{+} y(t)$ *and* $y(t) \rightarrow^{+} x(t)$.

Definition 10 (Form of a hierarchy) *A hierarchy is well-formed iff* $\forall x \in H,$ *there does not exist* $y, z \in H | y \geq x \,\Lambda\, z \geq x \,\Lambda\, y \# z$. *A process with an ill-formed hierarchy may block on some inputs.*

4.1.1.2 Compilability of Processes – Endochrony and Weak-Hierarchy

Endochrony of a process means that the process is able to read a flow of values irrespective of the time delays between subsequent values on the inputs, and still behaves deterministically. Endochrony is checked for different reasons – to ensure that a sequential implementation of a polychronous process can run deterministically in a latency insensitive manner – also to check if an asynchronous composition of processes will behave the same way as its synchronous composition, for instance read the values that are expected and emit them when they are needed. This second property is called isochrony between components and needs to be checked if the components are to be distributively deployed.

The definition of endochrony is expressed in terms of behaviors of processes. It is the property of a process that it is insensitive to the loss of timing relation between its inputs. Methods for checking endochrony are well studied [45], for example on the basis of theorems.

Definition 11 (Endochrony) *An endochronous process is a process that can reconstruct the timing relations of its signals from input streams of data with no timing relation [47].*

This property has been expressed in the formalism of clock hierarchies and data dependencies in the past.

Theorem 1 (Endochrony of a Process) *A process is endochronous iff it has a well-formed and acyclic clock hierarchy with a unique root. For proof, see [52].*

Another property of polychronous processes termed *weak hierarchy* has been defined in an attempt to extend the set of compilable processes by adding a class of constrained compositions of endochronous processes. Previously the weak-endochrony property [50] was defined that included endochronous processes and some of their compositions. However, weak-endochrony does not necessarily imply compilability. A weakly-endochronous process can be blocking, for instance, with a deadlock. The set of weakly hierarchical processes has been shown to sit between the set of endochronous processes and the set of weakly-endochronous processes [49].

Definition 12 (Weak hierarchy) *A process is weakly hierarchical if and only if either:*

- *It is endochronous;*
- *It is the composition of some weakly hierarchical processes such that the hierarchy* $\mathcal{H}$ *of the composition is well-formed and the dependency graph* $\mathcal{G}$ *is acyclic.*

Example 2 *Recall the process* `biadder` *from Figure* 1*. It may be decomposed into three processes, which can be proven endochronous (x1 is the root of P1, x2 the root of P2 and s the root of P3). Furthermore, it may be proven that their composition is well-formed and does not introduce cycles. Thus,* `biadder` *is weakly-hierarchic.*

```
Process biadder = (? integer x1, x2; ! integer y1, y2, s;)
(| P1 : (| y1 := y1$ init 0 +x1
         | s ^= (x1 = 0) |)
 | P2 : (| y2 := y2$ init 0 +x2
         | s ^= (x2 = 0) |)
 | P3 : (| s := ( y1 when s ) + (y2 when s) |)
 |)
```

Figure 3: Process biadder - a weakly hierarchical implementation

4.1.1.3 Non blockage of weak-hierarchic composition

Given the definition of weak hierarchy, a weakly hierarchic composition has to be non-blocking. This means that the components that are composed must be acyclic and have a well formed hierarchy. Since composition occurs on components that are already acyclic and well formed, the non-blocking checking can be done on the interface.

Lemma 1 *Knowing that P and Q are both weakly hierarchic, the condition of cyclicity can be checked only on the interface of P and Q.*

- *Let P and Q be two processes. The dependency graph of the interface of P and Q is called and the union of the transitive closures of the dependency graph of P and Q projected to their interface $P \cap Q$.*
- *If P and Q are acyclic process and the dependency graph of the interface of P and Q is acyclic then the composition $P|Q$ is acyclic.*
- *The composition of P and Q is cyclic if for any pair of signals x, y that belong to the interface $P \cap Q$, P holds a dependency between x and y and Q holds the opposed dependency.*

4.1.1.4 Clock Checking – Isochrony

Isochrony is the property that shows clock compatibility. Isochrony ensures that a system can be desynchronized – i.e., the composed components are tolerant to latency and can communicate through asynchronous channels without missing any communicated values, and result in deterministic execution.

Definition 13 (Isochrony) *A system of processes is isochronous if the asynchronous composition of the processes has the same behavior as their synchronous composition.*

The behaviors of two processes are considered the same if the sequences of data values on corresponding signals are the same given the same sequence of values on all inputs. This is called *flow equivalence* [46]. In other words, the information on alignment of the values to logical instants can be ignored while considering this equivalence. These have been considered in much detail in [46, 52].

Theorem 2 (Isochrony with Endochrony) *A system of processes is isochronous if the composition of the processes is endochronous.*

```
process endochronous_composition = (? C_1, C_2, C_s, x_1, x_2; ! integer s;)
(| s1 := P1(C_1,C_s,x1)
 | s2 := P2(C_2,C_s,x_2)
 | s  := P3(s1,s2)
 |) where s1, s2;
    process P1 = (? C_x, C_out, x; ! out)
    (| x ^= when C_x
     | y := y$ init 0 + x
     | out := y when C_out
     |) where y ;
    process P2 = (? C_x, C_out, x; ! out)
    (| x ^= when C_x
     | y := y$ init 0 + x
     | out := y when C_out
     |) where y ;
    process P3 = (? s_1, s_2; ! s)
    (| s := s_1 + s_2
    |)
end
```

Figure 4: Isochrony of an Endochronous composition

Figure 4 provides an example. Process endochronous_composition is isochronous because each of the composed processes is endochronous and the composition is itself endochronous. The main clock, which has to be faster than every input, and the actual presence of an input at this clock exist. P1 and P2 share the global clock, even if during some instants of this clock, when one input is absent, the process may not be working. Also note that the clock of P3 is "when C_s", which is different than the global clock. This is still correct because "when C_s" is a subclock of the global clock and the hierarchy of clock is respected.

This theorem has long been the rule in synchronous system. Proving that a system is endochronous is equivalent to finding a global clock of a system and relating every signal to this global clock. Unfortunately this can be very inefficient in heterogeneous systems where different components are paced at different clocks. In polychronous systems, which represent the best of these kinds of systems, the aim is to find a way to ensure isochrony of systems without synchronizing components to a global clock. To achieve the property of isochrony, weak-hierarchy is used.

Theorem 3 (Isochrony with Weak-Hierarchy) *A system of processes is isochronous if each process is weakly hierarchical and their synchronous composition is also weakly hierarchical.*

Proof (See [49]). The proof uses two properties of isochrony: (1) a non-blocking composition of two endochronous processes is isochronous and (2) a composition of processes in which every pair of processes is isochronous, is also isochronous. Since a weakly hierarchical process is a non-blocking composition of n-endochronous processes, each pair of processes forms an isochronous sub-system and the whole system is isochronous.

```
process weak_hierarchy_composition = (? C_s1, C_s2, x_1, x_2 ;! integer s;)
 (| s1 := P1(C_s1,x1)
  | s2 := P2(C_s2,x_2)
  | s ^= when C_s1 ^= when C_s2
  | s  := P3(s1,s2)
  |) where s1, s2;
 process P1 = (? C_out, x; ! out)
   (| y := y$ init 0 + x
    | out := y when C_out
    |) where y ;
 process P2 = (? C_out, x; ! out)
   (| y := y$ init 0 + x
    | out := y when C_out
    |) where y ;
 process P3 = (? s_1, s_2; ! s)
   (| s := s_1 + s_2
    |)
end
```
Figure 5: Isochrony of a weakly hierarchic composition

Figure 5 provides a second example of composition which is no longer endochronous despite the fact that each process of the composition is actually endochronous. The clocks of the process P1 and P2 can be different and unrelated. The clock of P3 is no longer a subclock of the global clock but it is still a subclock of P1's and P2's. The relation between the clock of P3 and the clocks of P1 and P2 is given by the equation "s $\hat{=}$ when C_s1 $\hat{=}$ when C_s2". The set of clock relation induces an order relation which is not total.

4.1.2 Develop methods for behavioral type extraction from MRICDF specification of a software component

4.1.2.1 Problems

1. Earlier *MRICDF* to *SIGNAL* code generation techniques treated function actors with a *C* definition as normal assignments. Also when the *SIGNAL* code was generated, – function call, interface details, function definition and the declaration were ignored. Any *MRICDF* model with a function actor was useless, except for direct *C* code generation. Conversion of *MRICDF* to *SIGNAL* was not able to be performed directly.

2. If one signal was branched out to multiple actors, the earlier way of handling conversion resulted in all signals being randomly assigned. For example, if signal *a* is an output of actor *A* coming from port *p*0 and is connected to input port *p*1 of actor *B*, input port *p*2 of actor *C* and input port *p*3 of actor *D*. Conversion assigns *p*1.*B*=*p*2.*C*, *p*2.*C*=*p*3.*D* and *p*1.*B*=*p*0.*A*, which is not correct. Because of this there are problems in calling *C* functions with arguments.

3. Because of these problems behavioral type extraction from MRICDF models with function actors are not able to be performed.

4.1.2.2 Solution

1. The assignment was corrected in conversion. Now *B*.*p*1=*p*0.*A*, *C*.*p*2=*p*0.*A* and *D*.*p*3=*p*0.*A* allowing function calls with arguments to be converted properly.

2. Generation of *SIGNAL* code from *MRICDF* is complete. It has been tested for the benchmark examples and works correctly.

3. Type information is able to be generated from the MRICDF models.

4.1.2.3 Anticipated problems and future work

1. Currently type extraction is done by converting MRICDF models to SIGNAL specifications and then using the "*–spec*" option for extracting the clock information regarding signals and sub-processes. The extracted information might contain a lot more information than what is needed. In the future, refining the extraction process to ignore detailed information will be explored.

2. A *C* function calling another *C* function is not currently handled. Each function actor can call only one *C* function. This might lead to restrictions in future designs but none have been noted as of now. The fix can be easy implemented in the future.

4.1.2.4 Extended Report: Develop methods for behavioral type extraction from MRICDF specification of a software component

Development of large software by means of a modular design approach often leads to long delays due to interface differences between interacting modules. Extracting behavioral type information from such software can provide us with valuable information to construct a formal foundation to investigate various compositional design methodologies between various interacting modules. Extraction of behavioral type information can be done at various stages, during the development process, or post-development during integration. This work focuses on how to extract behavioral type information during the development phase of the project. Once extracted, this information can be further used during integration to check the compatibility of various modules. Further, this work has been restricted to extracting behavioral information during the *EmCodeSyn* based high-level software synthesis methodology. Section 4.1.3 will discuss how to extract behavioral type information from *C* programs.

MRICDF is a data-flow specification. It does not store behavior type information. Hence it is necessary to transform it to another form, from which one can extract the behavior type information. To get into the required form, a source-to-source translation of the given MRICDF specifications to SIGNAL specifications is performed. All the primitive actors except function actor get translated to the corresponding actors in SIGNAL. The low level implementation details specified inside function actors are all abstracted and only the interface details are exported as the SIGNAL specifications. Since the functions are supposed to be executed in a single instant, just the interface details are sufficient to do the compatibility check. From the abstracted SIGNAL specifications, one can derive the clock relations, dependency information, and check for compatibility, causal loops, deadlocks, live-locks, race conditions etc. This methodology is illustrated with an example below.

Consider a simple producer-consumer system designed using *EmCodeSyn*. The data-flow model is shown in Figure 6. Function actors $F1$ and $F2$ represent producer and consumer modules respectively. Buffer actor $B1$ is used to provide feedback to function actor $F1$ and buffer actor $B2$ is used to tap the output of the consumer.

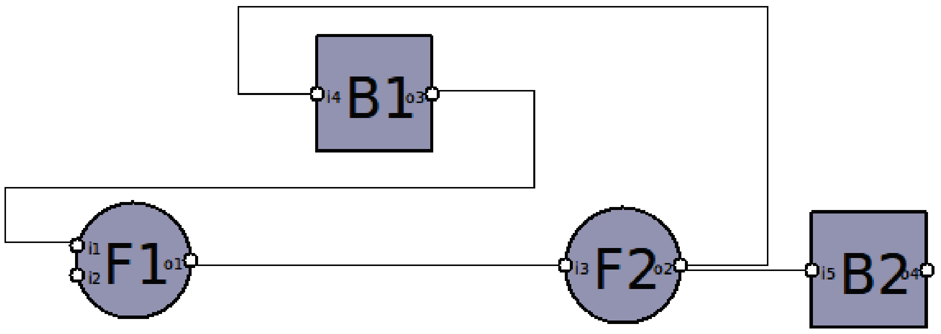

Figure 6: Producer-Consumer System

The actual behavior of the producer and consumer is specified using *C* function but the signals on which the producer produces inputs and the consumer consumes the inputs are specified externally. These control signals and their logic get exported as SIGNAL specifications but the implementation details of producer and consumer modules are hidden. The translated SIGNAL specification is shown in the listing of Figure 7.

```
process prodcons =
( ? boolean F1i2;! integer B2o4;)
(
| F1o1 := F1_prod1(F1i1, F1i2)
| F1i1 := B1o3
| F2o2 := F2_cons1(F2i3)
| F2i3 := F1o1
| B1o3 := B1i4 $ init 1
| B1i4 := F2o2
| B2o4 := B2i5 $ init 1
| B2i5 := F2o2
|)
where
integer F1i1, F2i3, B1i4, B2i5, F1o1, F2o2, B1o3;
function F1_prod1 =
( ? integer F1i1;
boolean F1i2;
! integer F1o1;
)
;
function F2_cons1 =
( ? integer F2i3;
! integer F2o2;
)
;
end % prodcons %;
```

Figure 7: Producer-Consumer System Translated SIGNAL Specification

To get the corresponding clock relations and dependency information between various signals and modules, one would compile the abstracted SIGNAL specifications using the *Polychrony* compiler with "*–spec*" option,. This information is shown in the listing of Figure 8. With this information one can perform checks for compatibility, causal loops, deadlocks, livelocks, race conditions etc.

```
process prodcons_POLY_TRA =
      ( ? boolean F1i2;
        ! integer B2o4;
      )
   pragmas
      Main
   end pragmas
   (| (| CLK_F1i2 := ^F1i2
       | CLK_F1i2 ^= F1i2 ^= B2o4
       | ACT_CLK_F1i2{}
       |) |)
   where
   event CLK_F1i2;
   process ACT_CLK_F1i2 =
         ( )
      (| CLK_F1i2 ^= F1o1 ^= F2o2 ^= B1o3
       | (| B2o4 := F2o2$1 init 1
          | B1o3 := F2o2$1 init 1
          | F1o1 := F1_prod1(B1o3,F1i2)
          | F2o2 := F2_cons1(F1o1)
          |)
       |)
      where
      integer F1o1, F2o2, B1o3;
      end
   %ACT_CLK_F1i2%;
   function F1_prod1 =
         ( ? integer B1o3;
             boolean F1i2;
           ! integer F1o1;
         )
      external
   %F1_prod1%;
   function F2_cons1 =
         ( ? integer F1o1;
           ! integer F2o2;
         )
      external
   %F2_cons1%;
   end
%prodcons_POLY_TRA%;
```

Figure 8: Producer-Consumer System Behavioral Information

4.1.2.5 Anticipated problems and Future work

Currently type extraction is performed by converting MRICDF models to SIGNAL specifications and then using "*–spec*" option for extracting the clock information regarding signals and sub-processes. The extracted information might contain a lot more information than what is needed. In the future, refinement of the extraction process to ignore detailed information will be investigated. A *C* function calling another *C* function is not handled currently. So each function actor can call only one *C* function. This might lead to design restrictions in future. The approach to use the *C–to–Signal* extraction tool to extract behaviors regarding internal function calls from top-level *C* routines will be investigated.

4.1.3 Develop methods for behavioral type extraction from C code

4.1.3.1 Work Completed

C code with the following constructs can be successfully converted to SIGNAL code using the C to SIGNAL (CTS) tool.

1. Assignments
2. If else
3. Nested if else
4. While loop

4.1.3.2 Anticipated problems and future work

1. SIGNAL code generated for "for" loops, "do...while", "if...else inside while" have to be tested.
2. Function calls inside a function have to be handled.

4.1.3.3 Extended Report: Automatic Conversion of C code to SIGNAL code

The C to SIGNAL conversion is achieved in two steps. The C code is first converted to its Static Single Assignment (SSA). Then the SSA is converted into SIGNAL by the CTS tool. SSA is an intermediate form where each variable is assigned exactly once. The SSA form is obtained from the C code with the help of GNU Compiler Collection (GCC).

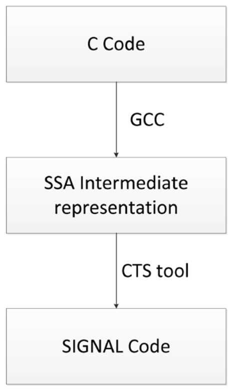

Figure 9: C to Signal Conversion Steps

4.1.3.3.1 C to SSA conversion

GCC generates the SSA code from the C code after some intermediate conversions. The GCC first converts the C code into Gimple Trees. The Gimple Trees are converted into a Control Flow Graph (CFG). The Control Flow graph is finally converted into SSA. This is done by GCC.

4.1.3.3.2 SSA to SIGNAL conversion

The generated SSA is parsed and relevant information is stored in a data structure. The block names, variables, constant variables, inputs, outputs, conditions and assignment statements in the SSA are parsed using regular expressions and stored in a data structure. The conversion of SSA to C code is described below:

4.1.3.3.3 SSA Blocks

The SSA consists of blocks which correspond to the various states in the program. Since all of the blocks consist of atomic statements with exactly one assignment for each variable, all of these statements can be executed in parallel. An enum type *lbl* is created in the SIGNAL code which takes names of all the blocks in the SSA. For example, if the SSA has blocks - <*bb* 2>, <*bb* 3>, <*bb* 4>, <*bb* 5>, the SIGNAL code is as follows:

```
type lbl = enum (bb2, bb3, bb4, bb5);
lbl label1, label_past;
```

label_past is the delayed version of label1. label1 and label_past are used to describe the state transitions which handle the control flow based on the conditions.

4.1.3.3.4 Assignment Statements

The assignment statements in SSA have the same form as in SIGNAL. Hence they are directly used from the SSA without any changes. An example of assignment statement in SSA is:

```
In C:      num = num + 2;
In SSA:    num_4 = num_1 + 2;
In SIGNAL: num_4:= num_1 + 2
```

4.1.3.3.5 PHI function

The PHI function is used to decide the final value of the variable from different versions of the variable based on the value of the basic block. The PHI functions are transformed to "when...default" statements in SIGNAL.

```
In SSA:    num_1 = PHI <num_5(4), num_6(5)>
In SIGNAL: num_1:= num_5 when (label1 = #bb4) default
                   num_6 when (label1 = #bb5) default num_1z
```

Here num_1z is the delayed version of num_1.

4.1.3.3.6 Conditional statements

The conditional statements specify the branching to a different block based on certain conditions. These are transformed to SIGNAL code using the enum *lbl*.

SSA:
```
<bb 3>:
  if (num_4(D) > 99)
    goto <bb 4>;
  else
    goto <bb 5>;
```

SIGNAL:
```
label1 := #bb4 when (label_past = #bb3)and(num_4buf > 99) default
         #bb5 when (label_past = #bb3)and(num_4buf <= 99) default
```

Here the buffered value of the input (num_4) is used. The buffered input is described in the next section.

4.1.3.3.7 Buffered values

The input variables and the variables in the PHI function need to be buffered in the SIGNAL code. For example, if int a_2 is the input and

```
num_1 = PHI <num_5(4), num_6(5)>
```

is the PHI function, the signal code for buffered values is as follows:

```
|a_2bufz := a_2buf $ init 0
|a_2buf  := a_2 when (label_past = #bb2) default a_2bufz

|num_1z := num_1 $ init 0;
```

The initial value for num_1 is a constant value if a constant value is assigned to it in the C code else it is 0.

4.1.3.3.8 Code Examples

Examples of C programs, their corresponding SSA and the automatically generated SIGNAL programs are given below.

4.1.3.3.8.1 If else

C Code
```
int ifElse(int flag1, int num)
{
if(flag1 == 0)
       num++;
else
       num--;
return num;
}
```

SSA form

```
int ifElse(int, int) (int flag1, int num)
{
 int D.1242;

<bb 2>:
 if (flag1_2(D) == 0)
    goto <bb 3>;
else
    goto <bb 4>;

<bb 3>:
 num_4 = num_3(D) + 1;
 goto <bb 5>;

<bb 4>:
 num_5 = num_3(D) + -1;

<bb 5>:
 # num_1 = PHI <num_4(3), num_5(4)>
 D.1732_6 = num_1;
 return D.1242_6;
}
```

SIGNAL code:

```
process ifElse = (? integer flag1_2, num_3;
!integer out1)
(| flag1_2bufz := flag1_2buf $ init 0
 | flag1_2buf := flag1_2 when (label_past = #bb2) default flag1_2bufz
 | num_3bufz := num_3buf $ init 0
 | num_3buf := num_3 when (label_past = #bb2) default num_3bufz
 | label_past := label1 $init #bb2
 | label1 :=
#bb3 when (label_past = #bb2) and (flag1_2buf = 0) default
#bb4 when (label_past = #bb2) and (flag1_2buf /= 0) default
#bb5 when (label_past = #bb3) or (label_past = #bb4) default #bb2
 | num_4 := num_3buf + 1
 | num_5 := num_3buf + -1
 | D01242_6 := num_1z
 | num_1z := num_1 $ init 0
 | num_1:= num_4 when (label1 = #bb3) default
        num_5 when (label1 = #bb4) default num_1z
 | out1 := D01242_6 when (label1 = #bb5)
 | flag1_2^= num_3^= when (label_past = #bb2)
 | num_1 ^= num_1z ^= label1 ^= label_past
 | num_3buf ^= flag1_2buf
 |)
where
integer num_4; integer num_5; integer D01242_6; integer num_1;
integer flag1_2buf, flag1_2bufz;
integer num_3buf, num_3bufz;
integer num_1z;
type lbl = enum (bb2, bb3, bb4, bb5);
lbl label1, label_past;
end;
```

4.1.3.3.8.2 Nested if else

C Code

```
int nestedIfElse(int flag1, int num)
{
    if(flag1==0)
   {
      if(num >= 100)
         num++;
     else
  num=num+2;
   }
   else
      num--;
   return num;
}
```

SSA representation:

```
;; Function int nestedIf(int, int)
int nestedIfElse(int, int) (int flag1, int num)
{
   int D.1735;

<bb 2>:
   if (flag1_3(D) == 0)
goto <bb 3>;
   else
goto <bb 7>;

<bb 3>:
   if (num_4(D) > 99)
goto <bb 4>;
   else
goto <bb 5>;

<bb 4>:
   num_5 = num_4(D) + 1;
   goto <bb 6>;

<bb 5>:
   num_6 = num_4(D) + 2;

<bb 6>:
   # num_1 = PHI <num_5(4), num_6(5)>
   goto <bb 8>;

<bb 7>:
   num_7 = num_4(D) + -1;

<bb 8>:
   # num_2 = PHI <num_1(6), num_7(7)>
   D.1735_8 = num_2;
   return D.1735_8;
}
```

SIGNAL Code

```
process nestedIfElse = (? integer flag1_3, num_4;
!integer out1)
```

```
(| flag1_3bufz := flag1_3buf $ init 0
 | flag1_3buf := flag1_3 when (label_past = #bb2) default flag1_3bufz
 | num_4bufz := num_4buf $ init 0
 | num_4buf := num_4 when (label_past = #bb2) default num_4bufz
 | label_past := label1 $init #bb2
 | label1 :=
#bb3 when (label_past = #bb2) and (flag1_3buf = 0) default
#bb4 when (label_past = #bb3) and (num_4buf > 99) default
#bb5 when (label_past = #bb3) and (num_4buf <= 99) default
#bb6 when (label_past = #bb4) or (label_past = #bb5) default
#bb7 when (label_past = #bb2) and (flag1_3buf /= 0) default
#bb8 when (label_past = #bb6) or (label_past = #bb7) default #bb2
 | num_5 := num_4buf + 1
 | num_6 := num_4buf + 2
 | num_7 := num_4buf + -1
 | D01245_8 := num_2z
 | num_1z := num_1 $ init 0
 | num_2z := num_2 $ init 0
 | num_1:=
     num_5 when (label1 = #bb4) default
        num_6 when (label1 = #bb5) default num_1z
 | num_2:=
     num_1 when (label1 = #bb6) default
        num_7 when (label1 = #bb7) default num_2z
 | out1 := D01245_8 when (label1 = #bb8)
 | flag1_3^= num_4^= when (label_past = #bb2)
 | num_1 ^= num_1z ^= num_2 ^= num_2z ^= label1 ^= label_past
 | flag1_3buf ^= num_4buf
 |)
where
  integer num_5; integer num_6; integer num_1; integer num_7;
  integer D01245_8; integer num_2;
  integer flag1_3buf, flag1_3bufz;
  integer num_4buf, num_4bufz;
  integer num_1z;
  integer num_2z;
  type lbl = enum (bb2, bb3, bb4, bb5, bb6, bb7, bb8);
  lbl label1, label_past;
end;
```

4.1.3.3.8.3 While loop

C Code

```
int whileCode(int n)
{
    int i=0;
    int result=1;
    while(i<3)
    {
       result=result*n;
       i++;
    }
    return result;
}
```

SSA form:

```
;; Function int cube(int) (_Z4cubei)
int whileCode(int) (int n)
{
   int result;
   int i;
   int D.1243;
   bool D.1733;

<bb 2>:
   i_3 = 0;
   result_4 = 1;
   goto <bb 4>;

<bb 3>:
   result_6 = result_2 * n_5(D);
   i_7 = i_1 + 1;

<bb 4>:
   # i_1 = PHI <i_3(2), i_7(3)>
   # result_2 = PHI <result_4(2), result_6(3)>
   D.1733_5 = i_1 <= 2;
   if (D.1733_5 != 0)
      goto <bb 3>;
   else
      goto <bb 5>;

<bb 5>:
    D.1243_9 = result_2;
    return D.1243_9;
}
```

SIGNAL Code

```
process whileCode = (? integer n_5;
       !integer out1)
(| n_5bufz := n_5buf $ init 0
  | n_5buf := n_5 when (label_past = #bb2) default n_5bufz
  | label_past := label1 $init #bb2
  | label1 :=
#bb3 when (label_past = #bb4) and (i_1z <= 2) default
#bb4 when (label_past = #bb2) or (label_past = #bb3) default
#bb5 when (label_past = #bb4) and (i_1z > 2) default
#bb2
  | result_6 := result_2z * n_5buf
  | i_7 := i_1z + 1
  | D01243_8 := result_2z
  | i_1z := i_1 $ init 0
  | result_2z := result_2 $ init 1
  | i_1:= i_3 when (label1 = #bb2) default
     i_7 when (label1 = #bb3) default i_1z
  | result_2:= result_4 when (label1 = #bb2) default
             result_6 when (label1 = #bb3)default          result_2z
  | out1 := D01243_8 when (label1 = #bb5)
  | n_5^= when (label_past = #bb2)
  | i_1 ^= i_1z ^= result_2 ^= result_2z ^= label1 ^= label_past
  |)
where
  integer  result_6;  integer  i_7;  integer  i_1;  integer  result_2;  integer
D01243_8;
  integer n_5buf, n_5bufz;
  integer i_1z;
  integer result_2z;
  type lbl = enum (bb2, bb3, bb4, bb5);
```

```
    lbl label1, label_past;
    constant integer i_3 = 0;
    constant integer result_4 = 1;
  end;
```

4.1.3.4 Extended Report: Automatic Conversion of C code to SIGNAL code, March 2012 update

4.1.3.4.1 Introduction

The C to SIGNAL tool which was developed, generated a long SIGNAL program with a large number of variables. With the goal to get a small SIGNAL program with few variables, the existing tool was modified. The first step involving converting the C code to its Static Single Assignment form remains the same. The second step of converting the SSA to SIGNAL has been changed to achieve an optimized SIGNAL program.

4.1.3.4.2 SSA To SIGNAL Conversion

The generated SSA is parsed and relevant information is stored. The way the SIGNAL program is generated from the information stored has changed.

4.1.3.4.3 SSA Blocks

The SSA blocks which correspond to the different states of the program remain the same. An additional <*bb* 0> state is added to the list of states. The <*bb* 0> state corresponds to initial state from where the program enters.

```
  type lbl = enum (bb0,bb2, bb3, bb4, bb5);
  lbl label1, label_past;
```

label_past is the delayed version of label1. label1 and label_past are used to describe the state transitions which handle the control flow based on the conditions.

```
  |label_past := label1 $init #bb0
```

4.1.3.4.4 PHI function

The PHI function is used to decide the final value of the variable based on the value of the basic block. The PHI functions are transformed to “when...default" statements in SIGNAL. The optimization in the SIGNAL code is achieved by substituting the variables from the assignment statements.

```
   |f := fNbuf when (label1 = #bb2) default
      (fz+ -10) when (label1 = #bb4) default
      (fz+10) when (label1 = #bb5) default
      fz
```

4.1.3.4.5 Conditional statements

The conditional statements have not been changed. They represent the branching to a different block based on certain conditions.

4.1.3.4.6 Code Example

An example C program, its corresponding SSA and the SIGNAL program are given below.

C Code

```
int whileIf (int n, int f)
{
    int i = 0;
    while(i<3)
    {
        if(f > 100)
        {
            n = n+2;
            f = f-10;
            i++;
        }
        else
        {
            n= n-2;
            f= f+10;
            i++;
        }
    }
    return n;
}
```

SSA Form

```
;; Function whileIf (whileIf)

whileIf (int n, int f)
{
  int i;
  int D.1256;

<bb 2>:
  i_4 = 0;
  goto <bb 6>;

<bb 3>:
  if (f_2 > 100)
    goto <bb 4>;
  else
    goto <bb 5>;

<bb 4>:
  n_7 = n_1 + 2;
  f_8 = f_2 + -10;
  i_9 = i_3 + 1;
  goto <bb 6>;

<bb 5>:
  n_10 = n_1 + -2;
  f_11 = f_2 + 10;
  i_12 = i_3 + 1;

<bb 6>:
  # n_1 = PHI <n_5(D)(2), n_7(4), n_10(5)>
  # f_2 = PHI <f_6(D)(2), f_8(4), f_11(5)>
  # i_3 = PHI <i_4(2), i_9(4), i_12(5)>
  if (i_3 <= 2)
    goto <bb 3>;
  else
    goto <bb 7>;

<bb 7>:
  D.1256_13 = n_1;
  return D.1256_13;

}
```

SIGNAL program

```
process whileIf = (? integer nN, fN; !integer out1)

(|nNbufz := nNbuf $ init 0
 |nNbuf := nN when (label1 = #bb2) default nNbufz
 |fNbufz := fNbuf $ init 0
 |fNbuf := fN when (label1 = #bb2) default fNbufz
 |label_past := label1 $init #bb0
 |label1 :=
#bb2 when (label_past = #bb0) default
#bb3 when  (label_past = #bb6) and (iz <= 2) default
#bb4 when  (label_past = #bb3) and (fz > 100) default
#bb5 when  (label_past = #bb3) and (fz <= 100) default
#bb6 when  (label_past = #bb2) or  (label_past = #bb4) or   (label_past = #bb5)
default
#bb7 when  (label_past = #bb6) and (iz > 2) default #bb0

 | i := iL when (label1 = #bb2) default
         (iz+1) when (label1 = #bb4) default
         (iz+1) when (label1 = #bb5) default
          iz
 | iz := i $ init 0

 |n := nNbuf when(label1 = #bb2) default
         (nz+2) when (label1 = #bb4) default
         (nz-2) when (label1 = #bb5) default
         nz
 |nz := n $ init 0

 |f := fNbuf when (label1 = #bb2) default
       (fz+ -10) when (label1 = #bb4) default
       (fz+10) when (label1 = #bb5) default
       fz
 |fz := f $ init 0

 |out1 := nz when (label1 = #bb7)

 |nN ^= fN ^= when (label1 = #bb2)
 |i ^= iz ^= n ^= nz ^= f ^= fz ^= label1 ^= label_past ^= nNbuf ^= fNbuf
 |)

where
    integer i, iz, n, nz, f, fz, D01246_13, nNbuf, nNbufz, fNbuf,   fNbufz;
    constant integer iL = 0;
    type lbl = enum(bb0, bb2, bb3, bb4, bb5, bb6, bb7);
    lbl label_past, label1;
end;
```

4.2 Development of Behavioral Type Inference Algorithm as a proof technique for Trustworthy composition

4.2.1 Reduction of behavioral type inference to Prime Implicate extraction problem.

Section 4.1 defined a weak-hierarchic process as the composition of endochronous processes. Using the code generation designed for endochronous processes and the decomposition strategy of this section to generate a composition of threads seems to be the right idea. This idea makes more sense in light of a trivial example, the parallel copy (|x:=a|y:=b|). The implementation can obviously be done using two threads and the absence of communication makes the endochronous code generation perfect for this translation. However, this approach brings two new problems: The first one is to find a good decomposition. Theory tells us that a composition of endochronous processes is weak-hierarchic and can be compiled, but given a weak hierarchic process how to find the good decomposition and the elemental pieces of endochronous code is an issue. The second problem is to manage the communication between the threads; shared variables need asynchronous synchronizations.

4.2.1.1 Semantic of the problem

4.2.1.1.1 Polychronous Elements

This section, introduces some formal definitions to precisely define the concepts of Polychrony. Informally, in a polychronous component, the clock of a signal is a sequence of logical instants at which the signal is computed or assigned new values. Not all signals at the interface are computed or assigned input values at every logical instant. A logical instant can be thought of as a maximal set of computation activities in reaction to one or more input changes.

This set of activities is maximal in the sense that any other activity would require another value to arrive on those inputs which triggered the current set of activities. Also, the computational activities in a reaction are partially ordered based on data dependency. If an activity produces data that another activity consumes immediately, they will be appropriately temporally ordered. However, from outside of the system, one cannot see the intermediate computations or interrupt them, until the reaction is completed. So the notion of a logical instant is not related directly to Newtonian time, but it denotes a set of activities triggered by one or more triggers, and it extends until a reaction is over. The next logical instant starts when another reaction is triggered.

Since not all signals participate in activities in all logical instants because their computation may be guarded by Boolean conditions, availability of inputs they react to, etc., the set of logical instants for each signal might be different. Also the polychronous model allows us to express concurrent computations; thus some of the activities may be concurrent and hence belong to a different sequence of reactions. For every signal – the set of logical instants (possibly infinite) that it participates in – is its clock. Thus signals may have different clocks; hence the model of computation is called polychronous or “multi-clocked”.

The signals at the interface therefore may be partitioned into classes and those who always participate in activities in the exact same logical instants are said to be synchronous with each other. Signals that participate in a subset of logical instants that another signal partakes in are said to be subclocks. Signals whose clocks are unrelated evolve asynchronously with each other. So one may find some signals always having new inputs arriving on them in the same logical instants, some signals have inputs arriving only in a subset of logical instants, and some might have inputs arriving in unrelated logical instants. All of these are captured in clock relations.

Definition 14 (Event) *Ξ is used to denote the set of all events. $\leq$ is a preorder on Ξ: $e \leq f$ means that e occurs before or concurrently with f. $\sim$ is the equivalence relation based on $\leq$: $e \sim f$ means that e and f occur simultaneously hence termed synchronous events.*

Definition 15 (Logical Instant or Instant) Υ *is used to denote the quotient of Ξ by $\sim$ as the set of instants. Thus a logical instant is a maximal set of events that are synchronous.*

Note that synchronous events may have a data dependency and hence may have a partial order inside the instant, different from ≤. This different order serves in the generation of sequential code which is not considered in this paper.

Definition 16 (Partial order on Instants) *$\Upsilon / \leq$ is the partial order on instants based on $\Xi / \leq$: $\forall a, b \in \Upsilon,\ a \leq b \Leftrightarrow \exists\, \alpha \in a,\ \beta \in b \mid \alpha \leq \beta$*

The above definition lifts the ≤ and the corresponding strict order < to the set of logical instants so one can compare the order of happening of some of the logical instants. The entities of the model are signals, defined on instants. A signal is a succession of values, happening at some specific instants. This specific succession of instants is the clock of the signal.

Definition 17 (Signal, Epoch and Clock) *Let T be a set of values – the type and $T_{\perp} = T \cup \{\perp\}$ its extension with a special value meant to represent absence, a* signal *is a function $\Upsilon \rightarrow T_{\perp}$ depending on a set of instants, its epoch.*

For a given signal, there exists one maximum set of instants $\gamma \subset \Upsilon$ such that γ is a total order in Υ and the signal associates a value from T to each instants of γ. Such a set is called the epoch of the signal, $\sigma(x)$ represents the epoch of the signal x.

Let us define the clock of a signal, a characteristic function that tells if a signal x is present or absent at any given instant t in Υ. Clock is a function $(\Upsilon \rightarrow T_{\perp}) \rightarrow \Upsilon \rightarrow \{true, \perp\}$ such that for a signal x returns another signal $(\hat{x})$ defined by: $\hat{x}(t)$ =true if $x(t) \in T$ and $\hat{x}(t) = \perp$ if $x(t) = \perp$

The notation $x(t_x^0)$ for the value for signal x at its initial logical instant is used. It is defined as such: $x(t_x^0) \in T$ and $\forall t \in \Upsilon, t < t_x^0 \Rightarrow x(t) = \perp$.

The notation $x(t_x^{-1})$ for the previous value of a signal at a given instant is used. It is defined as such: for all $t \in \Upsilon$ such that $x(t) \in T$ and $t = t_x^0$, yields $t_x^{-1} \in \Upsilon$ such that $x(t_x^{-1}) \in T$ and $\forall p \in Y, t_x^{-1} < p < t \Rightarrow x(p) = \perp$.

4.2.1.1.2 The Process and its logical Interpretations

Definition 18 (Dataflow Process) *A dataflow process P= (G, I, O) is defined by G, a directed graph in which edges are signals and nodes are dataflow actors, I is an inputs set and O is an outputs set; I and O are sets of signals. In the future, S will be defined as the set of signals of the process P, i.e. the set of edges of the graph G and E the set of epochs of every signal of S. Dataflow actors are either sub-processes (sG, I, O), either basic operators. Sub-processes are defined by a dataflow graph sG and have ports to access their input and output sets. These are called input ports and output ports.*

There are four basic operators which also have input and output ports and they are defined as follow:

- *A* function *(r = f(a,..,n)) calculates a result r from one or more operands a,..,n. The types of those signals depend on the function. The result is defined by r(t)=f(a(t),b(t)).*
- *A* delay *(r = a \$ init n) stores the current value of a and emits on r the previous stored value when a is present. On the first instant of a, the value n is emitted. The result is defined by* $r(t)=a(t_a^{-1})$ *when* $t>t_a^0$ *and r(t)=n when* $t=t_a^0$.
- *A* sampler *(r = a when b) transmits the values from a signal a to a signal r but only when the sampling signal b is true. The result is defined by r(t)=a(t) when b(t)=true and r(t)=⊥ when b(t)≠true.*
- *A* merge *(r = a default b) transmits to r the values from the signals a and b whenever they are present with a priority to b when both are present. The result is defined by r(t)=r(a) when r(a)≠⊥ and r(t)=r(b) when r(a)=⊥.*

The signals of the dataflow process connect zero or one output port and zero or one input port of any actor. Signals from the inputs set I necessarily connect no output port (undefined signals) and signals from the output set O necessarily connect one output port (define signals). Therefore, $I \cap O = \varnothing$.

Definition 19 (Epoch system of a Process)
From a Dataflow Process P, one may construct a Boolean Theory B_P *where axioms are equalities between Boolean formulas. Those formulas, which imply variables that represent epochs and Boolean conditions on signals, are defined by the basic dataflows operators as described in the second column of Figure 10. The idea behind this construction is that it can be used for a Prime Implicate (PI) minimization. The PI minimization produces a theorem where*

propositions are Boolean formulas in Conjunctive Normal Form and extract the essential Prime Implicates of the formulas.

Definition 20 (Well-clocked Process) *A process is said to be well clocked if the Boolean theory attached to it contains no contradiction.*

Example 5 Counterexample *y1 = 1 when x y2 = 1 when not x z = y1 + y2*
by1 = 1 and x=1 by2 = 1 and x=0 bz = by1 = by2 x = [x] ou [-x] [x] and [-x] = 0
models: (bx and [x])=(bx and [-x]) bx = false

Definition 21 (Dependency Graph of a process)
A dependency graph is a labeled directive graph where the nodes are signals and the edges represent dependencies relations between signals at some sets of instants. At each instant in a dataflow process, the events that are involved are ordered by an order relation: the dependency relation: $\rightharpoondown$. This relation means that one event can only happen after one other, for example if one signal y is a copy of the signal x, the two synchronous events y(t) and x(t) are connected by the relation $x \rightharpoondown y$. The relation $\rightharpoondown$ defines a graph for each instant. The relation $\to^{a}$ defined by $\forall x, y \in S, \forall a \subset Y, x \to^{a} y$, iff $\forall t \in a, x(t) \neg y(t)$ defines a graph for the whole process.

This graph can be directly constructed from the basic dataflow operators of the process, refering to the third column of the table shown in Figure 10.

Dataflow actor	Epoch / Boolean Relations	Data Dependencies
Function $r = a \star b$	$\sigma(a)=\sigma(b)=\sigma(r)$ $b_a=b_b=b_r$	$a \to \sigma(a)_r$ $b \to \sigma(b)_r$
Buffer $r=b\$$ *init n*	$\sigma(a)=\sigma(r)$ $b_a=b_r$	No dependency
Sampler $r=a$ when b	$\sigma(r)=\sigma(a)\cap\sigma(b=true)$ $b_r=b_a and b_{[b]}$, $b_b=b_{[b]} or [-b]$ $b_{[b]} and [-b]=false$	$a \to \sigma(r)_r$
Merge $r=a$ *default b*	$\sigma(r)=\sigma(a)\cup\sigma(b)$ $b_r=b_a or b_b$	$a \to \sigma(a)_\mathrm{r}$ $a \to \sigma(b)-\sigma(a)_r$

Figure 10: Formal definitions of Dataflow Actors

4.2.1.2 Clock tree decomposition using Prime Implicate theory

Given a theory of Boolean formulas that represent the clock relations of a specification, the master trigger is a signal whose clock is necessarily true whenever any other term is true. In an endochronous process it corresponds to a signal that has the greatest epoch. In a PI minimization, the variable that corresponds to a master trigger appears as a prime implicate since it can nullify the whole theory.

In a weak-hierarchic process, the set of epochs are not totally ordered and there is no master trigger. One can find a set of signals for which the sum of the epochs covers the set of instants of the process. Again, such a set of signals would appear in a PI minimization. These called partial triggers.

4.2.1.2.1 Finding the Master trigger of a specification

In [1], the authors show an algorithm to extract the master trigger of a process when it exists and prove that the existence of this signal is a necessary condition for sequential implementation of a specification. This condition implies that there is a signal that has events in every instant, thus the set of instants is totally ordered. It is related to the property of endochrony of a process.

Definition 22 (Master Trigger) *Let P be a well-clocked Dataflow Process and υ the set of instants of the process. Let x be a signal from I (the input set of P) with $\sigma(x)$ as its epoch and the properties that for each instant $t \in \upsilon$, t is in $\sigma(x)$, and there is no $s \in \upsilon$ for which the dependency graph of the process has a causal cycle. Then x will be termed the master trigger.*

This definition is deduced from the definition of endochrony that states that an endochronous process contains a master clock. A process which is endochronous can be sequentially implemented, as [1] proved that a process that has a master trigger is endochronous and can be sequentially implemented.

Theorem 4 (Existence of Master Trigger) *Given a well-clocked Dataflow Process P such that the dependency graph of P has no cycles, let B_P denote the system of Boolean equations derived from the actors. A signal x in I (the input set of P) is a master trigger if and only if its corresponding Boolean variable b_x in B_P has the property that if b_x is false, every other variable is false ($\forall b_y \in B_P, b_y b_x$).*

Proof: In Figure 10, the definition of epochs and Boolean Theory implies that the hypothesis $\sigma(y) \subset \sigma(x)$ and $b_y \Rightarrow b_x$ are equivalent.

Since the epoch of a master trigger *x* contains all instants of the process, for any signal *y* in P, then $\sigma(y) \subset \sigma(x)$. Thus $b_y \Rightarrow b_x$.

$\exists\ x \in P, \forall\ y \in P, b_y \Rightarrow b_x$ implies, $\exists\ x \in P, \forall\ y \in P, \sigma(y) \subset \sigma(x)$. Under the condition of no-dependencies, it also implies that x is a master trigger of P.

Theorem 4 can be exploited in two ways. First if a master trigger exists, a PI minimization of the Boolean equations would detect this master trigger as a positive literal among the Prime Implicates. Second, if a Prime Implicate is found and if the other criteria are met (no cycles in dependency graph and well-clocked), the process can be sequentially implemented.

4.2.1.2.2 Finding sets of partial triggers of a specification

Weakly-hierarchic processes were defined in [49]. These are an extension of endochronous processes. They do not aim to be implemented as sequential pieces of code but instead they can be implemented as a set of sequential threads. The endochronous processes are a special case where there is only one thread. From the definition, a weakly-hierarchic process is a composition of endochronous subprocesses. Thus, when an endochronous process has a master trigger, a weakly-hierarchic process has a set of partial triggers, each one having the greatest epoch among its own subset of signals.

Informally, the search is for a set of signals that act like master triggers of their own sub-processes (partition of process).

Definition 23 (Set of partial triggers) *A set of partial triggers is a set of signals such that when one of the partial triggers is absent, any of the others can be present but when all partial triggers are absent, all signals of the system are absent.*
Let P be a Dataflow Process and υ *the set of instants of the process. Let* $\{x_1,..x_n\}$ *be a set of signals from I with* $\sigma(x_1),..,\sigma(x_n)$ *as their epoch and the properties that*

- *for each pair* $x_i,x_j \in \{x_1,..x_n\}$, $\sigma(x_i) \subset \sigma(x_j)$,
- *for each signal* $y \in P$, *there is at least one trigger* $x \in \{x_1,..x_n\}$ *such that* $\sigma(x) \supset \sigma(y)$.
- *there is no* $s \in \upsilon$ *for which the dependency graph of the process has a causal cycle.*

Then $\{x_1,..x_n\}$ *will be termed the set of partial triggers for P.*

Theorem 5 (Weak Hierarchy of a process) *A process that has a set of partial trigger is weakly hierarchic.*

Proof: The authors of [53] define a composition of endochronous processes that is acyclic and well clocked to be weakly hierarchic.
Let P be a Dataflow Process which has a set of Partial Triggers.

For each signal $y \in P$, there is at least one trigger $x \in \{x_1,..x_n\}$ such that $\sigma(x) \supset \sigma(y)$. This describes a partition of the process in subprocesses. For each trigger x_i, there is a set of signals that form a subprocess P_i. The dependency graph of P has no cycle and the dependency graph of P_i is a subgraph of it. P_i is acyclic. P_i has a master trigger x_i; it is endochronous. P is by definition acyclic and well-clocked.

Theorem 6 (Discrimination of a set of Partial Trigger) *Given a well-clocked Dataflow Process P such that the dependency graph of P has no cycle, let* B_P *denote the system of Boolean equations* *derived from the actors. A set of signals $S=\{x_1..x_n\}$ is a set of partial triggers if and only if $\forall y \in P$, $\exists x \in S$ such that $y=0x=0$ and there is no pair $x_1,x_2 \in S$ such that $b_{x1}=0b_{x2}=0$.*

Proof: Like in theorem 4, this is the translation of the definition through the equivalency of $\sigma(y) \subset \sigma(x)$ and $b_y \Rightarrow b_x$.

Theorem 6 can be exploited the same way as theorem 4, i.e. a PI minimization of the Boolean Theory would detect a set of partial triggers and prove the implementability.

4.2.1.2.3 Scheduling the threads

Scheduling a thread involves strictly ordering events and refining two partial orders. One is based on definitions from the epoch system of the process and the other from the data dependency graph. The first order is extracted from the definitions of epochs in a structure called the Followers Set. The second order is used to reinforce the order between events of the same epoch based on the dependency graph.

Each trigger is the head of one of the main threads; this means each one of the partial triggers generates one thread. The event that happens at the epoch of the trigger has to happen at each instant. This set of events can be tied by data dependencies and the graph can be used to schedule them. For example, the simple process x=y has only two events $x(t)$ and $y(t)$ at the instant t. The epoch calculus and a trivial PI minimization tell us that x is the master trigger, and x and y have the same epoch. Then the graph dependency tells us that for any instant of $\sigma(x)$, there is a dependency $x \rightarrow y$. The thread will therefore begin with the event $x(t)$ by reading the value of the input x and then continue with the event $y(t)$ by attributing the value to the output y and emitting it. More likely, the thread has to manage several different epochs; those epochs are all subsets of the epoch of the trigger and can be ordered in a structure that will be called the Follower Set.

Definition 24 (Followers Set) *The Follower set of a trigger is a sequence of sets of epochs with the following property: in a Boolean Theory B_P of a process P, α belongs to the nth element of the Followers Set of a trigger x, if there is a formula in B_P that defines α with only epochs of lesser elements in this followers set.*

The first element of a Followers Set contains the epoch of the trigger. This is the first level of the thread. At this level, all the signals that share this epoch, 'the first element of the followers set', are known to be present and the epochs of the second level of the Followers Set can be evaluated.

Theorem 7 (Construction of the Follower set of an Endochronous Dataflow Process) *The construction of a follower set is also based on a Prime Implicate minimization.*

The follower set of the master trigger x_i is obtained by searching the Prime Implicant of successive boolean equations system B_M', B_M'', .. in which the previous elements are set to true.

The first element of this follower set, FS_1 is x_i, PI of B_M. The second element FS_2 is the set of PI of $B_M{}'=B_M\cup(\forall x\in FS_1, x=1)$, the third element is the set of PI of $B_M{}''=B_M{}'\cup(\forall x\in FS_2, x=1)$ and so on.

4.2.1.3 Shared Epoch

Communication in a multithreaded code generation requires synchronizations. In polychronous specifications, the synchronization is materialized by shared epochs, i.e. epochs that are subset of several roots. In a multithreaded implementation, the use of a synchronization barrier is needed to synchronize the shared epochs in order to protect the signals that are used in multiple threads.

Definition 25 (Shared Epoch)
Shared epochs in a weakly hierarchic specification are epochs of signals that can be accessed by several threads or calculations that use signals from several threads. Because each thread has its own main epoch and because a shared epoch is a subset of those epochs, it acts as a recurrent meeting between the threads. In an asynchronous implementation, this is materialized by a synchronization barrier. A variable that is present within a shared epoch is shared between the different threads that synchronize at this epoch. In particular, it can be defined with data from those threads.

The epoch of a shared variable is defined in the epoch of each thread that accesses it, and thus it should belong to the follower sets of the partial triggers of those threads. On the other hand, the algorithm described above rejects the shared epoch from their follower sets because when only one root is set to be present, shared epochs that belong to several partial triggers are necessarily absent. The basic detection of shared epochs requires building follower sets for each combination of roots, which is a waste because not all combinations own a shared variable. Use of a finer detection involving direct identification of the shared epochs and a generation of their follower sets is more efficient.

Definition 26 (Shared epochs) *Shared epochs are epochs that can be present only if at least two partial triggers are present.*
Because they synchronize several hierarchies, shared epochs have one definition in each of those hierarchies.
Property 1 (Identifications of shared epochs) *Consider shared epochs, clocks of signals defined by least two equations that individually appear in different follower sets.*

In practice, for each epoch defined by several equations, the root of each individual equation is extracted. If the partial triggers are different, the equations define a shared epoch if the roots are identical; the equations define a clock constraint in one of the follower sets, which may compromise the code generation.

Once shared epochs have been identified, it is possible to add in each concerned follower set a representative of this epoch and mark it as a synchronization point. One can then use each follower set to generate its own sequential code and place barrier synchronization at the marked epoch. This will create a set of threads that synchronize at each instant of the shared epoch.

4.2.1.4 Distribution of events into threads

Theorem 8 (Construction of the Follower sets of a Weakly hierarchical Dataflow Process) *The construction of a follower set for a weakly hierarchic process is close to the previous one except that each partial trigger and each shared variable has its own follower set. The follower set of the partial trigger x_i is obtained by searching the Prime Implicant of successive boolean equations system B_M', B_M'', .. in which the previous element is set to true and other roots are set to false.*

The first element of this follower set, FS_1 is x_i, PI of $B_M \cup (\forall x \in roots(M), x \neq x_1 x=0)$
The second element FS_2 is the set of PI of $B_M' = B_M \cup (\forall x \in FS_1, x=1)$, the third element is the set of PI of $B_M'' = B_M' \cup (\forall x \in FS_2, x=1)$ and so on.

The events of the process have to be distributed into several threads with respect to the different orders. The threads are synchronized with barrier, which means that the synchronization points have to be a minimum. The starting points of the threads are the partial triggers and the shared variables. Partial triggers are independent, they are not clock-related and the events at those epochs are not data-dependent with each other. The most efficient implementation is to have the set of events of each partial trigger in the same thread. Shared epochs, which are sub-clocks of several partial triggers, have their own threads and are synchronized with the main thread via a synchronization barrier.

Theorem 9 ensures that each event of the process belongs to exactly one thread.

Lemma 2 *All events belong to one epoch that is a subclock of a master trigger.*
If one event belongs to one epoch that is a subclock of several master triggers, this epoch is also a subclock of a shared epoch.

Theorem 9 (Placement of event in threads)
Let's suppose that M is a weakly hierarchical process with {Ci} its roots and {Csi} its shared epochs, define Ti as the thread attached to the partial trigger Ci and Tsi the thread attached to the shared variable Csi. Following the data dependency graph, every event that has no incoming dependency and belongs to the subclock of only one master trigger Ci are put in Ti. Events that have no incoming dependency and belong to a shared epoch Csi are put in Tsi. Events that have dependencies from only one thread are put in this same thread. Events that have dependencies from two threads, of two triggers Ci and Cj, must belong to one shared epoch Csi and are put in Tsi, behind a synchronization barrier. Events that have dependencies from two threads, one of trigger Ci and one of shared Csi, are put in the thread of the trigger, Ti, behind a synchronization barrier.

4.2.1.5 Algorithm

The code synthesis algorithm for Prime Implicate based sequential code generation is provided in Figure 11.

```
Types
 Epoch
 Epoch List
 Event      % a basic operation like read, write of IO or affectation of a variable
 Node      % a pair of one epoch and one event
 Node List
 Process    % contains Nodes
 BooleanTheory % Boolean theory of a process, contains Epochs and relations between
them
 DepGraph   % Dependency graph of a process, contains Nodes and relations between them
 Code      % A basic block of code
 Code List  % A Thread

Methods
       % returns the list of events at the epoch of epl
       Node List nodeAt(Epoch epl, Process p);

       % returns the list of nodes of G that have no precedent
       Node List headsOfGraph(DepGraph& pointer_g);

       % returns the list of PI of a BooleanTheory
       Epoch List primeImplicant(BooleanTheory b);

       % returns a new Boolean theory derived from b where e is set to True
       BooleanTheory setToOne(BooleanTheory b, Epoch List epl);

       % add items at the end of list

       x List addToList(x List list, x item);

       % remove e from g and link dependencies
       removefromGraph(depGraph& pointer_g, Event e);

       % returns the block of code of a node (if epoch then event)
       Code code(Node n);

codeGen(Process& pointer_p, Code List& pointer_t, DepGraph& pointer_g, BooleanTheory b)
{
       Epoch List epl = primeImplicant(b)
       Node List nl = intersection(nodeAt(epl,p),headsOfGraph(g))
       while(nl != null)
              foreach(Node n in nl)
              {
                          addToList(t, code(n))
                          remove(g,n)
                          nl = intersection(nodeAt(epl,p),headsOfGraph(g))
              }
       BooleanTheory b2 = setToOne(b,epl)
       codeGen(p, t, g, b2)

       nl = intersection(nodeAt(epl,p),headsOfGraph(g))
       while(nl != null)
              foreach(Node n in nl)
              {
                          addToList(t, code(n))
                          remove(g,n)
                          nl = intersection(nodeAt(epl,p),headsOfGraph(g))
              }
}
```

Figure 11: Recursive Algorithm for Prime Implicate based sequential code generation

Example 6 Code generation of a weakly endochronous process. *Here is an extract of generated multi-threaded code. The code contains three different threads: thread one and two are the main concurrent process of the program, thread_3 is the common subprocess, with the shared epoch. Synchronization is obtained with two barriers that protect the critical variable 'z'.*

```
#include <pthread.h>

pthread_barrier_t barr_1;
pthread_barrier_t barr_2;

int main()
{
    pthread_t thr_1;
    pthread_t thr_2;
    pthread_t thr_3;

    // Barrier initialization
    if(pthread_barrier_init(&barr_1, NULL, THREADS))
    {
      printf("Could not create barrier 1\n");
      return -1;
    }
    if(pthread_barrier_init(&barr_2, NULL, THREADS))
    {
      printf("Could not create barrier 2\n");
      return -1;
    }

    // threads creation
    if(pthread_create(&thr_1, NULL, &thread_1, NULL))
    {
      printf("Could not create thread 1\n");
      return -1;
    }
    if(pthread_create(&thr_2, NULL, &thread_2, NULL))
    {
      printf("Could not create thread 2\n");
      return -1;
    }
     if(pthread_create(&thr_3, NULL, &thread_3, NULL))
    {
      printf("Could not create thread 3\n");
      return -1;
    }
}

void * thread_1()
{
    boolean code;
    EXTRACT_1_initialize();
    while(code)
    {
      code = EXTRACT_1_step();
    }
    EXTRACT_1_Close();
}

boolean EXTRACT_1_step()
{
    if (!r_adder_a(&a)) return FALSE;
    x = a;
    w_adder_x(x);

    C_CLK_101 = (C_a ? (a == 0) : FALSE);

  ` if (C_CLK_101)
    {
      pthread_barrier_wait(barr_1);
      pthread_barrier_wait(barr_2);
    }
}

void * thread_2()
{
    boolean code;
    EXTRACT_2_initialize();
    while(code)
      {
        code = EXTRACT_2_step();
    }
    EXTRACT_2_Close();
}

boolean EXTRACT_2_step()
{
    if (!r_adder_b(&b)) return FALSE;
    y = b;
    w_adder_y(y);

    C_z = b == 0;
    if (C_z)
    {
      pthread_barrier_wait(barr_1);
      pthread_barrier_wait(barr_2);
    }
}

void * thread_3()
{
    while(code)
    {
        code = EXTRACT_3_step();
    }
}

boolean EXTRACT_3_step()
{
    pthread_barrier_wait(barr_1);
    z = x + y;
    w_adder_z(z);
    pthread_barrier_wait(barr_2);
}
```

4.2.1.6 Non blockage of weak-hierarchic composition

Given the definition of weak hierarchy, a weakly hierarchic composition has to be non-blocking. This means that the components that are composed must not contradict in signal definitions or add dependency cycles. This can be treated with a strong but sufficient property.

Theorem 10 (Strong condition for non blocking compositions of Hierarchies) *A composition of processes is non-blocking if for all set of k processes which share j variables, j<k.*

Proof: It is shown that both conditions of a blocking composition falsifies the theorem.

Let us suppose that a composition of processes has a cycle. There exists a minimal set of k processes that supports the cycle $k > 1$ because each process is acyclic. This cycle can be decomposed in j elemental dependencies with $j \geq k$ because each process supports a part of the cycle, and these dependencies involve exactly j variables (suppose that a variable can appear only once because dependency is a transitive relation and any loop can be reduced to a loop in which the variables are all different). The set of k processes shares j variables and $j < k$.

Proving well-formed hierarchy has a similar approach. Suppose that a composition of processes is ill-formed. This means that at least two variables x, y are exclusive and both super-clock of a third one z. x and y being exclusive means that there exists a w that is a super-clock of x and y. There are now four relations of inclusions involving four variables. Since each process is well formed, it means that those relations come from several processes (two, three or four). In the fewest possible cases, the number of shared variables is always at least equal to the number of processes.

Since the shared epochs are clearly identified by their equations, this property can be checked directly during code generation.

4.2.2 Investigate if SAT Modulo Theory (SMT) techniques will refine behavioral Types

4.2.2.1 Problems

1. Large scale examples will take lot of time if one uses entire MRICDF/SIGNAL specifications for verification of any property. As an example, causal loop detection was checked.

2. Identification of safe-operating area for a faulty system.

4.2.2.2 Investigated Methods

1. SMT Approach.

2. Polyhedra Approach.

4.2.2.3 Anticipated problems and future work

1. Investigated techniques will approximate any floating point data as integer data to the user defined precision. No open-source floating point polyhedral libraries exist as of now. Yices internally converts floating point to integer data. So high precision analysis might be problematic.

2. Investigated techniques will not work if there are non-linear constraints. Currently a non-linear constraint solver from MIT is being considered. No results as of now.

4.2.2.4 Extended Report: Investigate if SAT Modulo Theory (SMT) or other decision making techniques will refine behavioral Types

In Section 4.1.2 it was shown how to extract behavioral type information from the MRICDF models. After extraction of this information various kinds of analyses need to be performed such as causal loop detection, race condition check, and verification of other safety properties to ensure that the composition of the modules is compatible. The effort investigated the use of various decision-making tools to check these properties including SAT Modulo Theory (SMT), solvers, and later on Polyhedra libraries. An explanation of each work with their advantages and limitations is captured below.

4.2.2.5 SMT based safety property checking

This section shows how one can use SMT solvers for checking a particular safety property – causal loop detection. The approach is generic and can be used to verify most of the properties. In one of the earlier works [10], causal loop detection was done by generating SMT equations for the entire MRICDF model and this set of equations was given as an instance for the SMT solver. The disadvantage of [10] is that for a large scale example, the SMT instance will become huge and can lead to long running times – sometimes never ending. In this work, the first step is to mine the specifications for possible causal loops. The next step is to express the clock constraints of the dependencies as SMT equations and check if all the equations can be true at same time or not by evaluating the SMT instance. If the SMT instance evaluates to *true*, then there exists a causal loop, otherwise no.

An example will now illustrate this work. For the reason of expressiveness, Signal and *Polychrony* are used instead of MRICDF and *EmCodeSyn*. Consider the Signal code of Listing 1:

```
Listing 1: Constructive Causal Loop
process causal -smt =
(? integer initial , step 1, intMin ;
! integer min , avg , max;
)
(| initial ^= step 1 ^= intMin ;
|min := initial when intMin <5 default avg - step 1
|avg := min+ step 1 when intMin =10 default max - step 1
|max := avg + step 1 when intMin >10 default initial
|);
```

The code is compiled using *Polychrony* to check for the presence of possible causal loops. From the code in Listing 1, it is observed that when *isMin* is true, then *avg* depends on *max,* and *max* depends on *avg* causing a true causal loop. Similarly when *isMin* is false, then *avg* depends on *min,* and *min* depends on *avg* causing another true causal loop. Once the possible causal loops have been identified, the information regarding clock constraints is mined leading to the potential causal loops. After mining the clock relations are encoded as SMT equations and a SMT instance is constructed and tested for satisfiability. The clock relations are shown in Listing 2.

```
Listing 2: Clock relations
% Loop 1
(|{ avg --> max } when C_CLK _0
|{ max --> avg} when C_ CLK _1
|)
% Loop 2
(|{ min --> avg } when C_CLK _2
|{ avg --> min} when C_ CLK _3
|)
where , C CLK 0 := :(intMin = 10), C CLK 1 := intMin > 10,
   C CLK 2 := :(intMin < 5) and C CLK 3 := (intMin = 10)
```

Note that there are two sets of clock relations showing two possible true causal loops.

Any SMT constraint solver enriched with integer theories can be used. The latest YICES SMT solver [44] has been used as the constraint solver in this work. Translating the above clock relations as YICES input derives the equations in Listing 3.

```
Listing 3: SMT equations for Loop 1
;; Loop 1
( define intMin :: int)
( assert (and (not (= intMin 10)) (> intMin 10) ) )
( check )

Result :- sat (= intMin 11)

;; Loop 2
( define intMin :: int)
( assert (and (not (< intMin 5)) (= intMin 10) ) )
( check )

Result :- sat (= intMin 10)
```

Invoking YICES on these equations will give a SAT result as explained earlier. Also, YICES provides a counter example (*intMin*=11 & *intMin*=10) where the constraint is satisfied which matches with our earlier interpretation. Hence there exist true causal loops in the specification shown in Listing 1 and one possible way they can be formed is when *intMin*=11 & *intMin*=10. If YICES had given an UNSAT result, then one would conclude that the property is not satisfied and hence it is a false causal loop. Similarly any safety property can be expressed as an SMT instance and verified.

4.2.2.5.1 Limitations of this approach

Safety property verification such as causal loop detection is not a trivial problem. Given a data dependency loop, the complexity of checking if it is truly causal or not is at least NP-hard. If all inputs are Boolean signals, and the dependencies can be expressed as Boolean functions using ANDs and ORs and NOTs, then the problem would be the same as solving a SAT instance, which is NP-Complete. However, if the dependencies are able to be expressed as arbitrary functions over integers or reals or other complex data types, the problem is undecidable. This shows that any method used in verification must be based on heuristics and is likely not complete. One must strive for as close to a complete a solution as possible, while not compromising on the *soundness* of the solution. This is what has been done in this work. Another limitation of this work is that only non-floating point and linear constraints are handled. This results from YICES limitations, and is not a result of the approach taken here. Lastly, another shortcoming of this work is that if a property fails, the tool currently outputs only one of the many possible scenarios where the property will fail and not all of the scenarios where the property fails.

4.2.2.6 Polyhedra based safety property checking

This work tries to preserve the advantages of the SMT based approach and addresses minimizing its disadvantages. If the synthesized software has to interact with a physical environment, often additional range constraints on various inputs as well as outputs are provided. Analyzing the safety of execution often leads to analysis of reachability, invariants, and cyclic dependencies which may be affected by such range constraints. As explained in the last paragraph, while analyzing a specification for a safety property, even if it violates an invariant property, or shows cyclic dependency – even in a very limited area of its reachable state space - it will be rejected. For such specifications, instead of rejecting the specification outright, the synthesis tool should guide the user by showing the exact range of the input values (or equational relationships between the inputs as appropriate) by directing the resulting program to areas of the state space in violation. This is exactly the problem addressed in this section. To make decisions with range constraints, Polyhedral libraries are used as they can take affine relations as constraints. The example below will illustrate the problem being addressed by the work of this section. Consider the example shown in Listing 4.

Listing 4: Causal Loop Example

```
process AC_ DISPLAY = (? integer minT , curT , maxT ;
! integer disp _coldT , disp _hotT , disp _ normT )
(| minT ^= curT ^= maxT
| disp _ coldT := minT when curT <70 default curT
| disp _ normT := ( disp _ coldT +5) when curT =70 default
( disp _hotT -5)
| disp _ hotT := ( disp _ normT +5) when curT >80 default maxT
|);
```

A Boolean abstraction based check would replace each predicate by a Boolean variable taking arbitrary values, and will not consider the relationship between the predicates in their numerical domain. As a result, a causal dependency loop will be detected by such analysis because of the interdependency between *disp_normT* and *disp_hotT*. However, if the abstraction is cognizant of a theory of integers with ordering relations, then it would lower the Boolean abstraction to a model that considers intervals with ordering. On this model, one could prove that when *curT*>80, only then would such a causal dependency loop exist. Obviously, if this happens, the system will behave non-deterministically or will deadlock. If this information is explicitly presented to the user upon completion of the analysis, and the user can guarantee an additional input constraint, 70≤*curT*≤80, then generating code from this specification is completely legitimate, since the program will not display any deadlock behavior. In addition, if one wants to ensure safety, one could produce a wrapper that would intercept all inputs *curT* and check against this constraint, and filter out any occurrence of input values that violate the user guaranteed constraints. However, if the user can guarantee only 70≤*curT*≤90 the system will exhibit causal behavior when 80<*curT*≤90. But the system has a safe operating area, 70≤*curT*≤80. One could still apply a wrapper to prevent the system from moving outside its safe operating area, if it makes sense for the application.

This research proposes a polyhedral model based causality analysis technique which can accept Boolean, integer and rational input constraints and checks for violation of safety properties (e.g., existence of causal loops) in the constrained system. Based on polyhedral analysis of the constraints and specifications, a technique to identify the safe operating area of the system in terms of the bounds on the input and other linear constraints is also proposed. In the case of multiple safe operating areas, this technique lists all of them. Additionally, a safe code synthesis technique is proposed by adding wrappers to ensure that the resulting system does not behave non-deterministically or deadlock even when the input constraints are accidentally violated.

The proposed solution is illustrated with the example below. Consider the signal program shown in Listing 5, which is an extension of the program shown in Listing 4.

```
Listing 5: True Causal Loop
process AC_ DISPLAY = (? integer minT , curT , maxT , curP , curK
! integer disp _coldT , disp _hotT , disp _ normalT )
(| minT ^= curT ^= maxT ^= curP ^= curK
% Conditions %
| cond _1 := (( curT >= 2) and ( curT <= 18))
| cond _2 := (( curP >= 3) and ( curP <= 21))
| cond _3 := (( curK >= 25) and ( curK <= 35))
| cond _4 := (curT - curP >= -10)
| cond _5 := (( curT + curP >= 11) and ( curT + curP <= 33))
% Output Computation %
| disp _ coldT := minT when (curT < minT ) default curT
| disp _ normalT := ( disp _ coldT +10) when
(not( cond _1 and cond _2 and cond _3))
default ( disp _hotT -10)
| disp _ hotT := ( disp _ normalT +10) when ( cond _4 and cond _5)
default maxT
|)
where
boolean cond _1, cond _2, cond _3, cond _4, cond _5;
end;
```

When a Boolean abstraction is analyzed, it identifies the possibility of a causal loop because of the interdependency between *disp_hotT* and *disp_normalT* as shown in Listing 6.

```
Listing 6: Possible Causal Loop
(| { disp _ hotT --> disp _ normalT } when C_ CLK _31
| { disp _ normalT --> disp _ hotT } when C_CLK _23
|)
where , C_ CLK _31 = cond _4 and cond _5
C_CLK _23 = cond _1 and cond _2 and cond _3
```

One can invoke an SMT solver to check for nullity of clock constraints ($C\_CLK\_31 \wedge C\_CLK\_23$) on the path of the apparent loop. This is done by extracting the clock constraints and generating the predicates for the Yices SMT solver as shown in Listing 7.

```
Listing 7: Assertion in SMT solver and Solution
( define curT :: int) ( define curP :: int) ( define curK :: int)
( assert (and (<= curT 18) (<= curP 21) (<= curK 35)
(>= curT 2) (>= curP 3) (>= curK 25) (<= (+ curT curP ) 33)
(>= (- curT curP ) -10) (>= (+ curT curP ) 11) ) )
( check )

Result : SAT , Counter example : curT =8, curP =3, curK =25 %
```

Invoking the Yices solver will decide this condition as *satisfiable* (which indicates the existence of true causal loops) and it outputs *one* counter example to show a case where a causal loop may create a deadlock. If input constraints are included, an SMT solver will not be able to provide the safe operating region of the input space.

4.2.2.6.1 Constraint Extraction and Transformation for Polyhedral analysis

Given the input constraints shown in column 1 of Figure 12 for the SIGNAL program shown in Listing 4, the clock constraints for a possible causal loop are transformed to a system of affine inequalities and equations and are shown in column 2 of Figure 12. There exists an implicit logical intersection among all the constraints within each column of Figure 12. The constraints in Figure 12 need to be transformed into affine form to use the *PolyLib* library [51]. The system of translated affine inequalities is shown in Figure 13. This system is further abstracted to matrices before using Polylib APIs.

Input Constraints	Loop Constraints
$10 \leq \text{curT} \leq 40$ $10 \leq \text{curP} \leq 40$ $10 \leq \text{curK} \leq 40$	$2 \leq \text{curT} \leq 18$ $3 \leq \text{curP} \leq 21$ $25 \leq \text{curK} \leq 35$ $\text{curT} - \text{curP} \geq -10$ $11 \leq \text{curT} + \text{curP} \leq 33$

Figure 12: Input and True Causual Loop Constraints

Input	Loop
$\text{curT} - 10 \geq 0$ $-\text{curT} + 40 \geq 0$ $\text{curP} - 10 \geq 0$ $-\text{curP} + 40 \geq 0$ $\text{curK} - 10 \geq 0$ $-\text{curK} + 40 \geq 0$	$\text{curT} - 2 \geq 0$ $-\text{curT} + 18 \geq 0$ $\text{curP} - 3 \geq 0$ $-\text{curP} + 21 \geq 0$ $\text{curT} - \text{curP} + 10 \geq 0$ $\text{curT} + \text{curP} - 11 \geq 0$ $-\text{curT} - \text{curP} + 33 \geq 0$ $\text{curK} - 25 \geq 0$ $\text{curK} + 35 \geq 0$

Figure 13: Inequalities and Equations from Input and Loop Constraints

Figure 14 shows the plot of polyhedra representing both input constraint and true causal loop constraints. From the multiple views we see that there exists a region of intersection between the two polyhedra, which indicates the existence of true causal loops with the current input constraints.

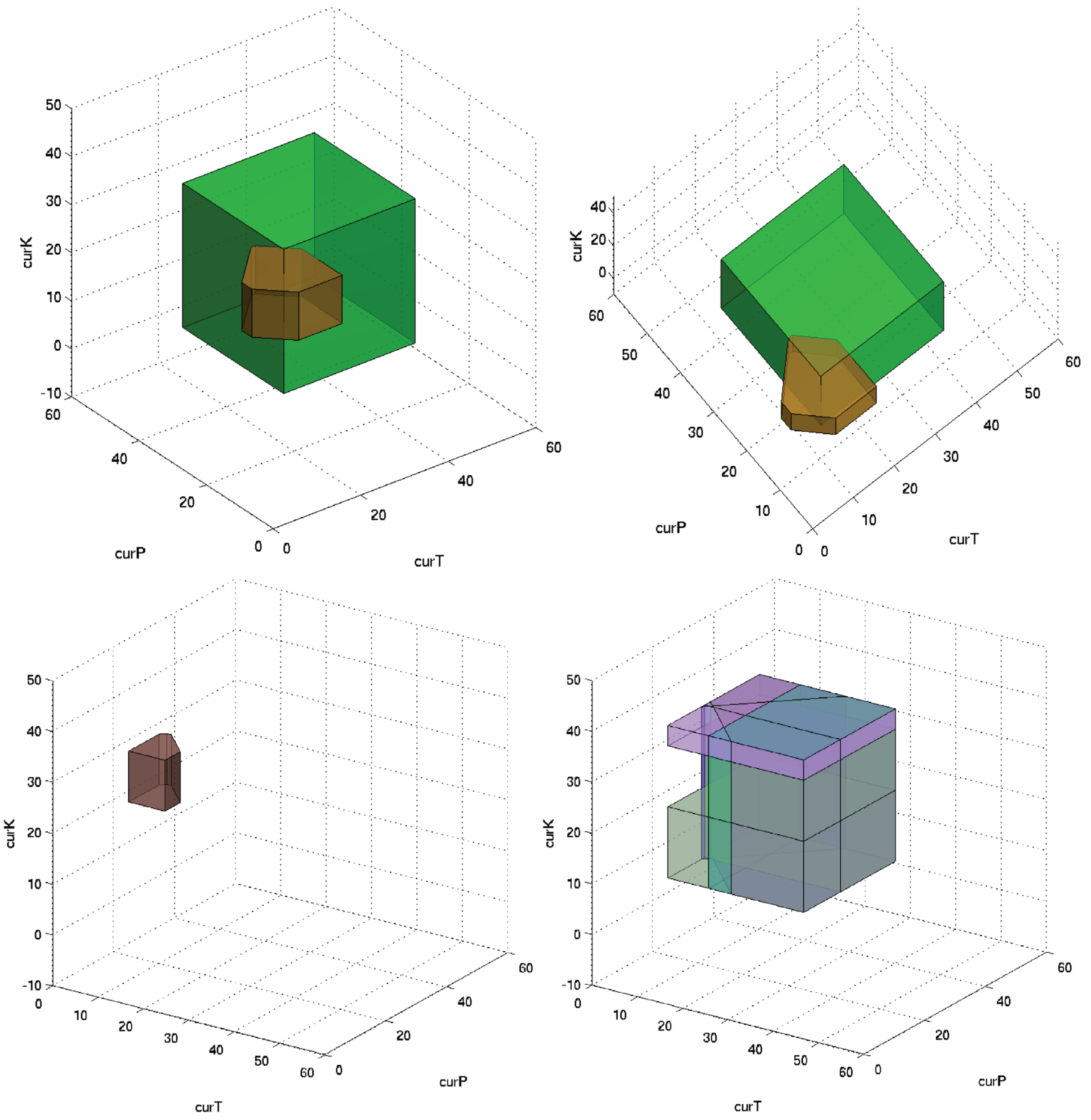

Figure 14: (Top) 3D-plot (multiple views) of Polyhedra representing Input and Loop Constraints. (Bottom) 3D plots of I ∩ L and I - L

4.2.2.6.2 Polyhedral Analysis

To obtain the bounds of the safe operating region and the region where the true causal loop exists, one applies two polyhedral operations from the *PolyLib* library.

i *DomainIntersection*(*I*,*L*): This operation returns the intersection of two polyhedral domains. This is used to compute $I \cap L$.

ii *DomainDifference*(*I*,*L*): This operation returns a new polyhedral domain which is the difference, $I - L$.

Both of these operations may return many sub-polyhedra instead of one single resultant polyhedron. The union of all of the sub-polyhedra will yield the resultant polyhedron. Figure 14 also shows the plots for both $I \cap L$ and $I - L$ respectively. Note that the plot of $I - L$ actually is the union of six different polyhedra.

4.2.2.6.3 Limitation of Polyhedral libraries

Almost all of the existing polyhedral libraries including the one used here, *PolyLib*, have restrictions in that they can only accept integer constraints. In the proposed technique, all rational constraints are multiplied by the least common multiple to obtain integers, and any floating point numbers are truncated based on the precision specified by the user. The truncated floating point constraint is then multiplied by a suitable number such that it becomes an integer.

4.2.2.6.4 Safe code synthesis using Wrapper

From the result of polyhedral analysis, the bounds on inputs for the safe operating region are obtained and must be checked before actually passing them to the process, so that the process remains in safe trajectories. Wrapper code is inserted which prevents any inputs violating the conditions of safety from being passed forward. The user of the synthesis tool is given the option to choose if such implementation makes sense in the application domain. Listing 8 shows the wrapped code for the SIGNAL program shown in Listing 4.

Listing 8: Signal program of Listing 7 with wrappers

```
process AC_ DISPLAY = (? integer minT , curT , maxT ;
! integer disp _coldT , disp _hotT , disp _ normT )
(| minT ^= curT ^= maxT ^= cond _1
| cond _1 := (( curT >= 70) and ( curT <= 80))
| disp _ coldT := ( minT when curT <70 default curT ) when cond _1
default DEFAULT _ VALUE
| disp _ normT := ( ( disp _ coldT +5) when curT =70 default
( disp _hotT -5) ) when cond _1
default DEFAULT _ VALUE
| disp _ hotT := (( disp _ normT +5) when curT >80 default maxT )
when cond _1 default DEFAULT _ VALUE
|)
where
bool cond _1;
end;
```

4.3 Tool development work

Developing large case studies using *EmCodeSyn* is difficult because of current GUI limitations of the EmCodeSyn tool. Below are some of the changes that have been implemented to mitigate this problem.

4.3.1 EmCodeSyn improvement

Current status: The actor shapes have been changed, so that now each actor has a different shape. The earlier zoom feature used to zoom in/out of the entire canvas, but changes have now been implemented to the GUI such that each actor can be individually zoomed in or out. There was a problem with the canvas and scroll bars for large MRICDF models which has now been fixed allowing the scrollbar to adapt dynamically. Originally all the lines were 1-D lines and because of this the models were not presentable as the lines were allowed to pass over another actor causing a messy display. An automated graph layout mechanism has been implemented which performs orthogonal layout of MRICDF models based on the Open Graph Drawing Framework (OGDF)[54]. It has been completely integrated into *EmCodeSyn* for orthogonal layout and a demonstration of the clean display of crossing lines has been performed.

4.3.1.1 Simcdf: Simulink models to MRICDF import

Current Status: The Simcdf tool can generate MRICDF XML files from multi-layer Simulink MDL files. Currently all of the blocks in MDL files are considered as function actors. On the other hand writing *C* function definitions for most of Simulink blocks for code generation has been completed. Integration of Simdcf to *EmCodeSyn* is complete and a demonstration has been performed. *C* code generation from the Simulink models can now be done in *EmCodeSyn*.

Further improvements: The current Simcdf tool accepts a subset of Simulink models. It is planned to extend the Simcdf tool to also accept integrators, differentiators, and other timing sensitive blocks.

4.3.1.2 Sigcdf: SIGNAL to MRICDF import

Future work: SIGNAL examples can be leveraged not only in the distribution but to build MRICDF models from them. This is useful in two ways. First is for verification of the EmCodeSyn compiler implementation and second is to compare the quality of code generated by Polychrony as compared to EmCodeSyn. Apart from these is the added advantage of ready-made MRICDF models, although this will require the ability to parse the entire SIGNAL grammar.

4.3.2 CTS Tool

Section 4.1.3 describes the C to SIGNAL conversion tool.

4.4 Implementation of Behavioral type interference algorithm

Section 4.1.2 describes the behavioral type extraction algorithm.

4.5 Development of Prime Implicate extraction algorithms

4.5.1 Introduction

The terminology used here for logical formulas is standard: An *atom* is a propositional variable, a *literal* is an atom or the negation of an atom, and a *clause* is a disjunction of literals. An *implicate* of a logical formula is a clause entailed by the formula. Thus a clause C is an implicate of a logical formula $\mathcal{F}$ if C is satisfied by every interpretation that satisfies $\mathcal{F}$. An implicate is a *prime implicate,* if it is not a tautology and if no proper subset is an implicate. Prime implicates are useful in the analysis of Boolean formulas (logical formulas) derived from polychronous (MRICDF) specifications, and the primary goal of the State University of New York at Albany team is the development of effective techniques and systems for computing the prime implicates of those systems.

Techniques that scale are crucial because real systems will generate large formulas, and the general problem of finding all prime implicates of a logical formula is *NP*-hard. The project has produced advances that enabled significant speedups, even for the general prime implicate problem, and especially for Boolean theories arising from MRICDF specifications.

4.5.2 Computational Advances

4.5.2.1 Leaving the Original Algorithm

In [34], a branch-by-branch analysis leads to the algorithm introduced there. In [35], a set oriented characterization is not only more intuitive but leads to a more efficient version of the algorithm. The added efficiency stems from three improvements.

- First, many subsumption checks required by the original algorithm are revealed by the set oriented analysis to be unnecessary and are avoided in the improved algorithm.
- A second improvement results from avoiding tries in which branches have distinguishing marks, necessarily stored at the ends. Checking the marks entails traversing the branch and is almost as expensive as a subsumption check. Instead, identically typed branches are kept in single tries.
- Realizing clause-based operations recursively on entire sets, represented as tries, provides a third improvement. Experiments indicate that the trie-based operations outperform branch-by-branch operations, and that the advantage increases with the size of the trie.

The difference in performance of the newer algorithm over the original is quite dramatic:

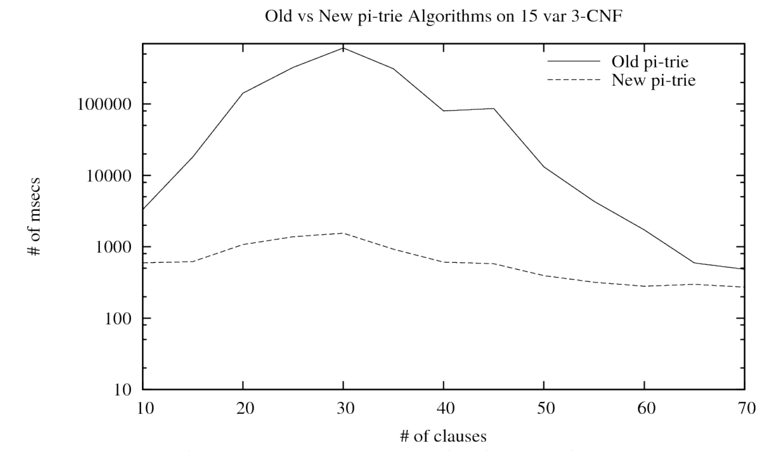

Figure 15: Old vs New pi-trie algorithm

Figure 15 compares (in log scale) the *pi*-trie algorithm from [34] to the updated version in [35], using the recursive, trie-based and operators. The input for both algorithms is a 15-variable 3-CNF with varying numbers of clauses and the runtimes averaged over 20 trials.

4.5.2.2 Filters and Search Space Reduction

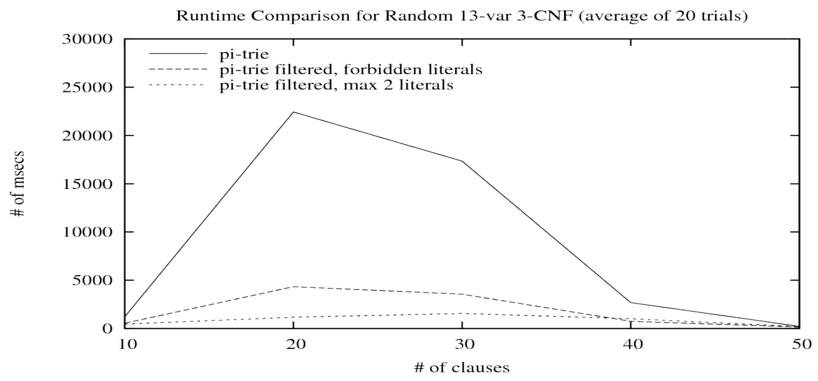

Figure 16: pi-trie Filtering

In synthesis of polychronus systems, prime implicates that are positive, containing only positive literals, and short, especially unit prime implicates, are of special interest. Such prime implicates can always be selected from the entire set of prime implicates, but generating only the prime implicates of interest is not only preferable, but much more efficient [35, 36]. Figure 16 has the results of an experiment with a 13-variable 3-CNF formula. Two filters are used: the first is "max length 2," and the second excludes clauses containing any of the literals v_3, v_5, v_6, or $\neg v_7$. The algorithm's singular design removes clauses not satisfying the specified filter from the computation itself, not just from the results, thus reducing the entire search space.

4.5.2.3 Decomposition

The goal of decomposition is to reduce a single computation on a formula to be *independent* computations on subformulas. For prime implicates, as well as for many other problems, this can only be accomplished if the subformulas are variable-disjoint.

The *pi*-trie algorithm creates conjoined subformulas via variable substitution. Decomposition analysis can enable a choice of variable orderings for the algorithm, so that conjoined variable disjoint subformulas (of roughly equal size) result.

The system developed here is equipped with a preprocessing component that selects a variable ordering for the algorithm that induces a favorable decomposition. Variable disjoint subformulas are detected and analyzed independently, thus reducing the computational load of the system.

Each of the seven examples listed in Figure 17 was run in eight different trials. In the first four trials, the variables were reordered in a way that favors decomposition, while no reordering was done in trials 5-8. Within these groups of four, the *pi*-trie algorithm was set up with the following option choices.

1. A plain vanilla *pi*-trie algorithm set to build the entire *pi*-trie.

2. A *pi*-trie algorithm filtered for positive prime implicates only.

3. A plain vanilla *pi*-trie algorithm to build the entire *pi*-trie, but recognize variable-disjoint subformulas.

4. A *pi*-trie algorithm filtered for positive prime implicates only, and recognizing variable-disjoint subformulas.

Problem	ReOrd Vars	ReOrd Time	# Orig Vars	# Actual Vars	# Orig Clauses	# Actual Clauses	Avg_Time In Msecs	Constructor Options
BlackBoard_1.sat	yes	37	118	54	243	112	timeout	none
BlackBoard_1.sat	yes	37	118	54	243	112	4788.0	(pos only)
BlackBoard_1.sat	yes	37	118	54	243	112	timeout	(decomp.)
BlackBoard_1.sat	yes	37	118	54	243	112	854.0	(pos only)(decomp.)
BlackBoard_1.sat	no	0	118	54	243	112	timeout	none
BlackBoard_1.sat	no	0	118	54	243	112	2633.0	(pos only)
BlackBoard_1.sat	no	0	118	54	243	112	timeout	(decomp.)
BlackBoard_1.sat	no	0	118	54	243	112	1188.0	(pos only)(decomp.)
FWS_1.sat	yes	3	48	17	97	31	66.0	none
FWS_1.sat	yes	3	48	17	97	31	17.0	(pos only)
FWS_1.sat	yes	3	48	17	97	31	57.0	(decomp.)
FWS_1.sat	yes	3	48	17	97	31	11.0	(pos only)(decomp.)
FWS_1.sat	no	0	48	17	97	31	71.0	none
FWS_1.sat	no	0	48	17	97	31	18.0	(pos only)
FWS_1.sat	no	0	48	17	97	31	153.0	(decomp.)
FWS_1.sat	no	0	48	17	97	31	35.0	(pos only)(decomp.)
GCD_1.sat	yes	5	55	20	125	47	37.0	none
GCD_1.sat	yes	5	55	20	125	47	10.0	(pos only)
GCD_1.sat	yes	5	55	20	125	47	28.0	(decomp.)
GCD_1.sat	yes	5	55	20	125	47	7.0	(pos only)(decomp.)
GCD_1.sat	no	0	55	20	125	47	46.0	none
GCD_1.sat	no	0	55	20	125	47	23.0	(pos only)
GCD_1.sat	no	0	55	20	125	47	56.0	(decomp.)
GCD_1.sat	no	0	55	20	125	47	24.0	(pos only)(decomp.)
pEHBH_1.sat	yes	3	48	14	110	30	34.0	none
pEHBH_1.sat	yes	3	48	14	110	30	12.0	(pos only)
pEHBH_1.sat	yes	3	48	14	110	30	23.0	(decomp.)
pEHBH_1.sat	yes	3	48	14	110	30	5.0	(pos only)(decomp.)
pEHBH_1.sat	no	0	48	14	110	30	23.0	none
pEHBH_1.sat	no	0	48	14	110	30	5.0	(pos only)
pEHBH_1.sat	no	0	48	14	110	30	20.0	(decomp.)
pEHBH_1.sat	no	0	48	14	110	30	5.0	(pos only)(decomp.)
prod_con_prim_1.s	yes	8	76	39	163	81	12715.0	none
prod_con_prim_1.s	yes	8	76	39	163	81	1440.0	(pos only)
prod_con_prim_1.s	yes	8	76	39	163	81	10386.0	(decomp.)
prod_con_prim_1.s	yes	8	76	39	163	81	95.0	(pos only)(decomp.)
prod_con_prim_1.s	no	0	76	39	163	81	7059.0	none
prod_con_prim_1.s	no	0	76	39	163	81	1006.0	(pos only)
prod_con_prim_1.s	no	0	76	39	163	81	6982.0	(decomp.)
prod_con_prim_1.s	no	0	76	39	163	81	633.0	(pos only)(decomp.)
resetcounter_1.sat	yes	2	26	13	51	24	13.0	none
resetcounter_1.sat	yes	2	26	13	51	24	3.0	(pos only)
resetcounter_1.sat	yes	2	26	13	51	24	12.0	(decomp.)
resetcounter_1.sat	yes	2	26	13	51	24	2.0	(pos only)(decomp.)
resetcounter_1.sat	no	0	26	13	51	24	14.0	none
resetcounter_1.sat	no	0	26	13	51	24	3.0	(pos only)
resetcounter_1.sat	no	0	26	13	51	24	19.0	(decomp.)
resetcounter_1.sat	no	0	26	13	51	24	6.0	(pos only)(decomp.)
watchdog_1.sat	yes	5	59	28	119	55	726.0	none
watchdog_1.sat	yes	5	59	28	119	55	181.0	(pos only)
watchdog_1.sat	yes	5	59	28	119	55	474.0	(decomp.)
watchdog_1.sat	yes	5	59	28	119	55	51.0	(pos only)(decomp.)
watchdog_1.sat	no	0	59	28	119	55	808.0	none
watchdog_1.sat	no	0	59	28	119	55	147.0	(pos only)
watchdog_1.sat	no	0	59	28	119	55	701.0	(decomp.)
watchdog_1.sat	no	0	59	28	119	55	58.0	(pos only)(decomp.)

Figure 17: Results of 5 technique combinations for solving the Blackboard problem

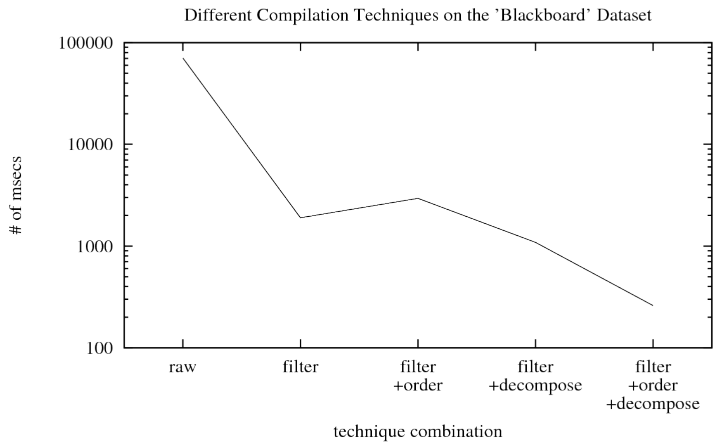

Figure 18: The Blackboard Problem

The Blackboard problem is the largest and most difficult of the example problems. Figure 18 shows more recent runs on the Blackboard problem without timeout limits and using a progression of five technique combinations. Figure 17 shows the "timeout" on all four options not filtering for positive prime implicates.

The first technique, discovery of equivalent variables, is applied in all cases of Figure 17. A preprocessor searches the given formula for simple variable equivalences and then collapses the equivalence classes into singletons.

The next technique applied is filtering. This alone increases efficiency by more than an order of magnitude. This is followed by ordering. By itself, ordering does not help at all, but a significant speedup results from the synergistic combination of ordering and decomposition.

In summary, on the Blackboard problem, the raw *pi*-trie algorithm requires about 1.6 minutes with discovery of equivalent variables. But with filtering, reordering, and decomposition, the positive prime implicates are obtained in about 0.3 seconds.

4.5.2.4 Dynamic Programming

The main routine of the *pi*-trie algorithm searches recursively through the space of assignments by incrementally substituting truth constants for variables in the formula. This may result in independent subproblems (as discussed in the previous section). But in addition, identical subproblems may arise along different paths in the recursion. For example, let $F = \{\{a, b, c, \neg d\}, \{c, \neg d, e, f\}\}$. Then each of the following substitutions for c, d, e *and* f yield the formula $\{\{a, b\}\}$:

c	D	e	f
0	1	0	1
0	1	1	0
0	1	1	1

When *c*=0 and *d*=1, the resulting clauses $\{\{a,\ b\},\ \{e,\ f\}\}$ are variable-disjoint, and thus any assignment satisfying the clause $\{e, f\}$ simply removes it from the formula without affecting the rest. Repeated subproblems produced by such variable-disjointness are recognizable and strategies can be developed via dynamic programming to prevent repeated work. A version of the *pi*-trie system that employs this capability has been developed. While there is considerable overhead, significant speedups are likely to occur when the problems get quite large and have small treewidth. However, large MRICDF systems appear to induce (large) Boolean theories with small treewidth and may therefore be more amenable to this technique.

4.5.2.5 Graph-Based Analysis of MRICDF Theories

The actors of an MRICDF system yield clauses of the corresponding Boolean theory as follows:

The epochs of a **function** or of a **buffer** are identical; each pair a, b yields the clauses – $\{\overline{a}, b\}$, $\{\overline{b}, a\}$. The union of the epochs of a **priority merge** yield the following clauses – $\{i_1, i_2, \overline{o}\}$, $\{\overline{i_1}, o\}$, $\{\overline{i_2}, o\}$. The epochs of a **sampler** lead to union and intersection clauses – $\{\overline{c}, [c], [\neg c]\}$, $\{\overline{i}, \overline{[c]}, o\}$, $\{\overline{[c]}, \overline{[\neg c]}\}$, $\{\overline{[c]}, c\}$, $\{\overline{[\neg c]}, c\}$.

Variables that are equivalent, representing signals with the same epoch, can be treated as identical, substantially reducing the size of the formula.

Typically two-clauses contain one positive and one negative literal and arise from epoch subsets. The subset relation partially orders $\mathcal{I}$, the *system epoch*, i.e., the set of all instants. This partial order enables representation of $\mathcal{I}$ as a dag (*directed acyclic graph*) $\mathcal{D}$. The union of the maximal elements of $\mathcal{D}$ is all of $\mathcal{I}$, and thus the set $\mathcal{M}$ of maximal elements represent a positive implicate, though not necessarily a prime implicate. However, any minimal subset of $\mathcal{M}$ whose union is the system epoch is a prime implicate. These prime implicates are especially useful for this project.

There are well known fast (i.e., polynomial) algorithms that find the set of maximal elements of a dag. A fast algorithm that removes unnecessary maximal elements, tentatively called the *union algorithm*, has been developed, and a prototype system that implements the union algorithm is under development. The prototype was able to find a prime implicate subset of $\mathcal{M}$ for each of the seven examples in Figure 17.

However, the union algorithm does not always produce a prime implicate subset of the set of maximal elements. Counterexamples have been discovered, but all seem to be rather unusual cases such as MRICDF systems.

Even though the output is not guaranteed to be a prime implicate, it is expected to be useful, both in its own right and as a filter for the *pi*-trie system. Focusing on this specialized version of the prime implicate problem may enable analysis of systems far too large to be handled by even the most advanced prime implicate algorithms.

4.6 Case Study

Purpose: Currently, the extraction tool, CTS, is not ready to automatically generate code for full-scale systems. In order to provide some preliminary results this research has manually converted two large-scale control systems from C to SIGNAL. The objectives of this study are two-fold. First, by converting the systems to SIGNAL, it is possible to illustrate the capabilities of SIGNAL to detect behavioral conflicts that would otherwise go undetected by a C compiler. Second, the converted systems will be able to serve as a baseline for future automatic translation attempts to be compared against.

4.6.1 Case Study Examples

4.6.1.1 ArduPilot System Description

Ardupilot is an open source hardware and software platform that was developed as a part of the Arduino open-source electronics prototyping platform. The Ardupilot system consists of the hardware which is placed on an Unmanned Aerial Vehicle (UAV) and a radio controller that is held by the user to control the UAV. The basic components of the hardware inside the UAV are the following:

- RC receiver
- Global Positioning System (GPS) receiver
- Servo Motors - for controlling the direction of the UAV
- Various sensors including position and pressure
- Ardupilot Board - board containing the controller of the UAV

The radio controller in the hand of the user is used to communicate the roll, pitch and the throttle information to the UAV. The firmware for the system is written in the Arduino language, which is an extension of Embedded C.

The following is a description of the modes in which the Ardupilot works.

- Mode1: Manual Flight – The UAV is controlled completely using the radio controller.
- Mode2: Stabilize – The radio controller is used to control the UAV. However, if the user does not use the radio controller, the UAV will automatically stabilize.
- Mode3: Fly-by-wire-A – The UAV will automatically go to the programmed point controlling the altitude and speed.
- Mode4: Fly-by-wire-B – The UAV will automatically go to the programmed point controlling the altitude. The airspeed is controlled manually in this mode.
- RTL: Return to launch mode in which UAV will return to the programmed launch point and will circle until manual control is established. The UAV can be nudged in this mode.

- Loiter: The aircraft will circle around the current position. The UAV can be nudged in this mode.

4.6.1.2 Ardupilot Software

The Ardupilot case study analyses the software running on the microcontroller of the Ardupilot system. It tries to explain the cases where there could be problems if the system were implemented in C and whether these issues would be detected earlier if SIGNAL were to be used to develop the software of this embedded system. There are various modules implemented in the Ardupilot software. A brief description of the modules is provided in Figure 19.

Module Description	
Name of Module	Major Functions of the Module
Waypoint	Responsible for reading different waypoints from EEPROM and updating waypoint information
Timing	Contains timer functions
System	Configures the pin modes , reading and writing values from EEPROM, setting the mode of the Ardupilot by changing mux setting and failsafe operation
Servo	Setting servo muxes to switch between auto and manual mode, adjusting servo positions
Sensors	Reads values from analog inputs to determine the value of roll, pitch and speed
Radio	Initializes the radio and reads radio value
Print	Prints various variables out through the serial port
Navigation	Calculates the roll and pitch of the ardupilot that have to be set to navigate to the set destination waypoint
GPS	GPS related functions
Events	Switches mode based on various events that happen in the system
Control Navigation	Navigation helper functions
Control Attitude	Sets the integrator values to maintain the desired pitch and roll calculated by control navigation module
Attitude	Sets the position of the servo based on the calculated pitch and roll
Ardupilot	The main module

Figure 19: Ardupilot software – module descriptions

The main module of the Ardupilot software, apart from variable initialization and initial configuration set-up, consists of an infinite loop (*while(true){…}*) that is responsible for running the Ardupilot. This infinite loop can be compared to a process in SIGNAL. Figure 20 provides the list of major function calls in this main while loop and a description of the function.

Functions in the Main Loop	
Function Name	Description
read_control_switch	Read 3-position switch on radio
read_radio	Filters radio input
throttle_failsafe	Checks for throttle failsafe condition
read_XY_analogs	Read IR sensors values
decode_gps	Read in the GPS position
reset_location	Reset waypoint to the starting point
read_analogs	Read analog inputs
stabilize	Function that helps in stabilizing the UAV
navigate	Navigation control loop that helps in navigating from one waypoint to another
update_throttle	Updates the throttle value

Figure 20: Ardupilot software – functions in main loop

4.6.1.3 Case study examples on Ardupilot

Why was Ardupilot chosen for the Case Study?

Ardupilot is a real-time embedded system where safety of the system is critical. Even a small deviation from an expected behavior can result in a large amount of deviation in the UAV's expected path and can even lead to loss of the UAV. The software on the Ardupilot consists of various modules that are integrated together. This provides an opportunity to check if there can be errors in the software because the behavior of the module is not captured.

Ardupilot: Case1 -

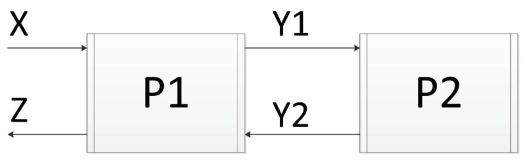

Figure 21: Case 1 Block Diagram

Figure 21 represents a sub-process (P2) being called by a main process (P1). This case shows that even if the sub-process functionality is implemented correctly, it produces non-deterministic output when integrated into the main process. This is because the main process expects the clock of x and z to remain the same, i.e. for every input x there has to be an output z generated. The sub-process however takes an input only after it produces the output and this behavioral property of the sub-process makes it incompatible for integration with the main process.

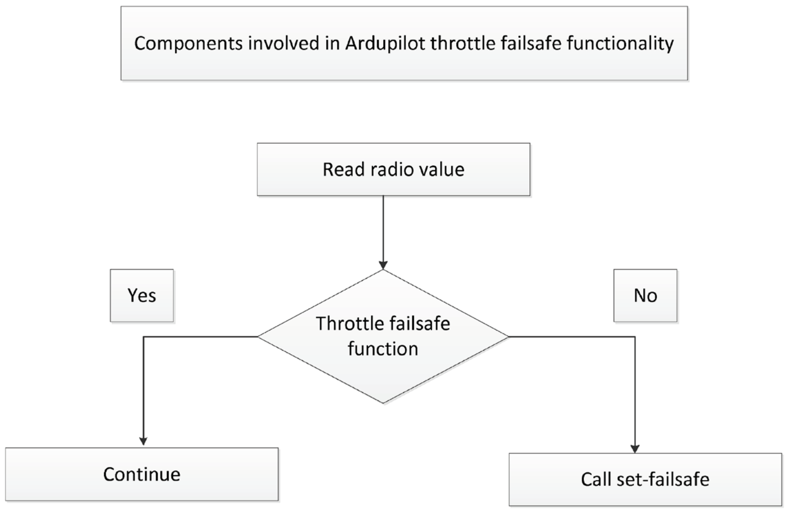
Components involved in Ardupilot throttle failsafe functionality
Read radio value
Yes
Throttle failsafe function
No
Continue
Call set-failsafe

Figure 22: Case 1 – Ardupilot Block Diagram

In the case of Ardupilot, the main loop consists of read_radio function that reads a channel value from the radio that corresponds to the throttle, and the throttle_failsafe function that checks if the value read is within acceptable limits, as shown in Figure 22. The SIGNAL program, as shown in Listing 9, shows a throttle function where the output clock does not match the input clock. This occurs because the throttle function takes four iterations to compute the output. This means that if this behavior is not taken into consideration while designing the system, it will result in an undesired output.

Listing 9: SIGNAL program of throttle function

```
process case1 =( ? dreal radio_ch3;! boolean throttle_failure;)
(| radio_ch3 ^= throttle_failure
%radio input processing %
| radio_out := 0.9 * radio_ch3

% throttle failsafe function %
| throttle_failure := (throttle (radio_out) > 100.0)
|)
where
dreal radio_out;

      process  throttle =
      ( ? dreal in1;
      ! dreal out1;
      )
      (| cnt := 4 when (cntz = 0) default (cntz - 1)
      | cntz := cnt $ init 0
      | inbuf := in1 when (cnt = 4) default inbufz
      | inbufz := inbuf $init 0
      | calc := (inbuf + calcz) when (cnt >  0) default 0.0
      | calcz := calc $ init 0
      | out1 := calcz when (cnt = 0)
      | in1 ^= when (cntz = 0)
      | calc ^= inbuf ^= cnt
      |)
      where
      integer cnt, cntz;
      dreal inbuf, inbufz, calc, calcz;
      end;

end
```

The SIGNAL code of Listing 9 models this scenario. The clock calculus of SIGNAL is able to identify the problem and indicates that there is a clock constraint. The clock constraint implies that this system is behaviorally incompatible. Thus, the block 'throttle' has to be redefined to be behaviorally compatible with the system. However, the same system implemented in C will result in non-deterministic output.

Ardupilot: Case2 -

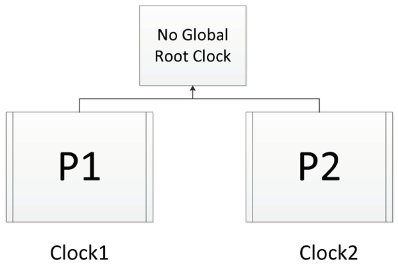

Figure 23: Case 2 Block Diagram

In the Figure 23 there are two processes. Process 1 takes an input and generates the sum from 1 to the value of the input. Process 2 takes an input and multiples the value by 4. Both of these processes could represent two separate components developed independently. They will function correctly when they are tested. However, integration of both of the processes shown in Listing 10 will result in an error.

```
  Listing 10:
  process update_throttle  = ( ? integer ch3; ! integer throttle)
  (| throttle := sum (ch3) + multiply (ch3)
  |)
  Where

        process sum = ( ?  integer in1; ! integer out1; )
        (| j ^= cnt
         | in1 ^= when (cntz = 0)
         | cntz := cnt $ init 0
         | cnt := in1 when ( cntz = 0) default (cntz-1)
         | jz := j $ init 0
         | j := (jz + cnt) when (cnt > 0) default 0
         | out1 := jz when (cnt = 0)
         |)
        where
        integer cnt, cntz, j, jz;
        end;

        process  multiply =
        ( ? integer in1;
          ! integer out1;
        )
        (| cnt := 4 when (cntz = 0) default (cntz - 1)
         | cntz := cnt $ init 0
         | inbuf := in1 when (cnt = 4) default inbufz
         | inbufz := inbuf $init 0
         | calc := (inbuf + calcz) when (cnt >  0) default 0
         | calcz := calc $ init 0
         | out1 := calcz when (cnt = 0)
         | in1 ^= when (cntz = 0)
         | calc ^= inbuf ^= cnt
        |)
        where
        integer cnt, cntz;
        integer inbuf, inbufz, calc, calcz;
        end;

  end
```

The SIGNAL code in Listing 10 was able to identify that the two components are incompatible because it was able to extract the root clock for the 'update_throttle' process. The addition operator requires both the operands to have the same clock. However, the two processes have different clocks and hence the system cannot have a root clock generated.

4.6.2 EmCodeSyn as a Code Generation Tool for an UAV

4.6.2.1 Introduction

Ardupilot is an open source hardware and software platform. The hardware board used consists of an ATmega2560, a 16-bit micro-controller. The board is fully programmable and with the help of a GPS module and Inertial Measurement Unit (IMU) sensors it can be used to develop an Unmanned Aerial Vehicle. Software for the Ardupilot can be programmed using the Arduino Programming language. The language is similar to C and includes constructs that can be used in the programming of the micro-controller. The software can be written to make the UAV work in various modes that can include Autopilot, Manual, and Circling. The code generation tool is aimed at generating code for various functions of the UAV using EmCodeSyn. EmCodeSyn is a framework of code synthesis from a multi-rate data flow based specification. It is a synchronous programming tool based on the Multi-rate Instantaneous Channel Connected Data Flow formalism.

4.6.2.2 Hardware Components

The hardware components of the UAV are the following:

1. The Ardupilot Mega Micro-controller Board

2. The IMU Sensor Board

3. The Radio Control Transmitter and Receiver

4. The GPS Module

4.6.2.2.1 ArdupilotMega (APM) Micro-controller Board

The micro-controller board is the component that contains the ATmega2560 micro-controller. The software written is programmed into the Atmega2560. The micro-controller runs at 16 MHz and has 12 timers, 256KB Flash, 4KB EEPROM and 8 KB RAM. It also has 16 10-bit ADC channels, four 8-bit PWM channels and four programmable serial Universal Synchronous-Asynchronous Receiver/Transmitter's (USART). The picture of the board is shown in Figure 24. This board also has eight input and eight output ports that can be used to receive and transmit Pulse Width Modulated (PWM) signals. The Global Positioning System (GPS) port on the board can be used to interface with the GPS module.

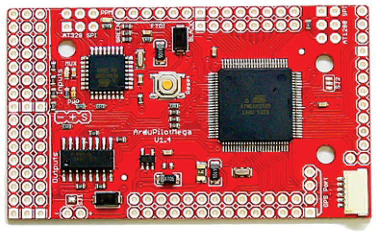

Figure 24: ArdupilotMega Micro-controller Board

4.6.2.2.2 IMU Sensor Board

This sensor board has various sensors that are used to measure different parameters of the UAV. The data of the sensors in this board is used to access the speed and orientation of the UAV and hence can be used to fly the UAV in autopilot mode. The main sensors on the board are a 3-axis accelerometer and a 3-axis gyroscope that output data with the help of a 12-bit analog-to-digital converter (ADC). The accelerometer measures the inertial forces that are exerted on the UAV. The gyroscope measures the rotation of the object attached to it in the X, Y, and Z co-ordinates. Both the data from the accelerometer and gyroscope are used in estimating the orientation of UAV in space. The board also has pressure sensors that measure the altitude of the UAV.

A mini Universal Serial Bus (USB) port on the board can be used to interface the board with a computer. This port is used in programming the APM board and is also used in sending data to a computer during testing and for log collection. This board has ports that are used for connecting it with the APM micro-controller board. The sensor data during flight is sent via these ports to the micro-controller. The picture of the sensor board can be found in Figure 25.

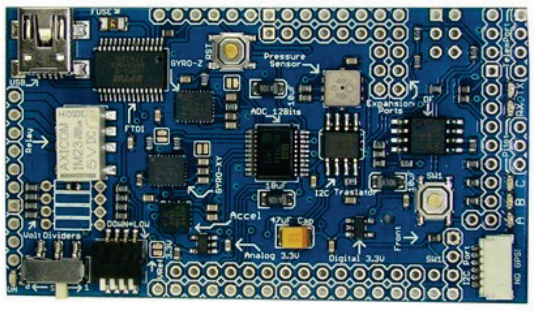

Figure 25: IMU Sensor Board

4.6.2.2.3 Radio Control Transmitter and Receiver

The Radio Control (RC) Transmitter and Receiver enable communication with the ground station and the UAV. A nine Channel Turnigy transmitter and receiver are used. The transmitter has two control sticks with each stick capable of moving in two directions. These four directions can be programmed to transmit data about the throttle, yaw, pitch, and roll that the user wants the UAV to have from the ground station. Each of the four parameters sends PWM signals in separate channels. Figure 26 shows the transmitter and the four parameters that can be communicated using the control sticks.

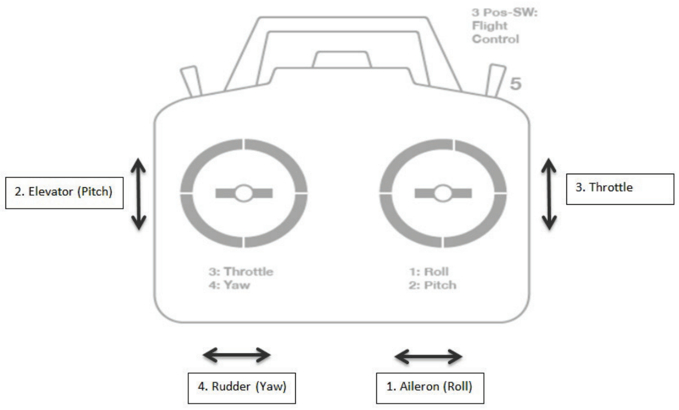

Figure 26: RC Transmitter Schematic

Two switches on the transmitter are used to communicate the flight mode information. One of the switches is a 3-way switch and the other one a 2-way switch. Hence, up to six modes can be programmed for the UAV flight. The pulse widths emitted by each of the switch positions has to be programmed in the transmitter. One of the channels of the transmitter is chosen (Channel 5) and each position of the switch is programmed to output a specific PWM. The six PWM's programmed are 1165, 1295, 1425, 1555 and 1815. Figure 27 shows the picture of the transmitter.

Figure 27: RC Transmitter

The RC receiver has to be coupled with the transmitter first. It is placed inside the UAV along with the APM board. The receiver has nine channels that are directly coupled with the transmitter. Each of these channels will give a PWM signal as its output. These channels are connected with the input channels on the APM board. The RC receiver is powered by a 5 V battery. The APM board receives its power from the RC receiver, shown in Figure 28.

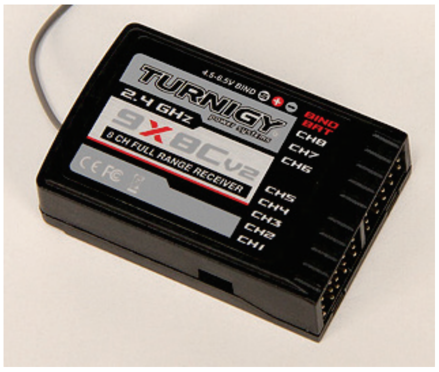

Figure 28: RC Receiver

4.6.2.3 Work Progress

All of the hardware required for the UAV has been set-up and the functioning of the APM board has been verified using Hardware-in-the-loop simulation. The software structure of the Ardupilot has been covered in the previous section. Work on Code generation is ongoing. The aim of the Code generation tool is to write all the important functions of Ardupilot in EmCodeSyn and to be able to implement various modes on the UAV.

4.6.3 AutoSAR

4.6.3.1 System Description

AUTOSAR, the automotive open system architecture, is a layered software architecture developed jointly within the automotive industry to create an open standardized interface for automotive hardware. The architecture is designed to interface with electronic control units, ECUs, that employ real-time operating systems and interact strongly with hardware. Specifically, it interacts with systems built around 16 or 32-bit microprocessors that can communicate on the controller area network (CAN), local interconnect network (LIN), or FlexRay protocols. Additionally, it is designed to be extensible so drivers can be added for future devices.

As previously mentioned, AUTOSAR is a layered system. The lowest layer is the microcontroller abstraction layer, which contains internal drivers for accessing internal peripherals and memory mapped devices of the microcontroller. Above this is the engine control unit (ECU) abstraction layer, which interfaces with the underlying layer while also providing drivers for external peripherals connected to the microcontroller. The top layer is the services layer. This layer provides operating system functionality, network and memory services, and manages the ECU state.

4.6.3.2 Arctic Core Software

Arctic Core is an open-source implementation of the AUTOSAR standard. For the purpose of this example, the microcontroller layer is focused on, since it is the most fundamental layer and runs closely atop the hardware. A brief description of the core modules of this layer is provided in the table below.

AutoSAR Microcontroller Modules	
Name of Module	*Major Functions of the Module*
CPU	This module is responsible for providing communication protocols to read and set the myriad on-chip peripherals
MCU	Initializes or De-initializes the microcontroller
OS	Contains the methods for the running operating system to interact with the controller
EVENT	Provides system with external event handling capabilities
COUNTER	Tracks internal time for synchronizations and alarms
MEMORY	Interfaces with on-chip memory
TASK	Provides methods for manipulating and interacting with running tasks
PCB	Implements the Process Control Block
SWAP	Allows for OS context swapping
ARCH	Responsible for maintaining and tracking the system stack
ALARM	Implements timed alarms for the system
COM-INTERNAL	Implementation of internal communication protocols

Figure 29: AutoSAR Microcontroller Modules

All the source C files for a PowerPC microcontroller kernel and supplemental driver libraries from the Arctic Core project were manually translated to the SIGNAL language. First, the required new type definitions from these modules were identified and consolidated in a SIGNAL module (modules in SIGNAL are roughly equivalent to a C library). Then, each module was translated into a corresponding SIGNAL module while preserving their original behavior. Lower level functions were sometimes problematic because SIGNAL does not have a notion of pointers. To overcome this obstacle, some functions utilize SIGNAL's ability to embed C source code. The depth of timing analysis that can be performed on such functions is limited because the embedded C code is treated as a black box by the SIGNAL compiler. However, the timing of the process call and the interface variables for these functions still contribute to the overall system model. Finally, the individual modules were composed into a single top-level module, representing the entire system.

4.6.3.3 Arctic Core Example

Using this translated system module an example execution was reconstructed from the original project.

```
process test_driver = ( ? event start; )
(| btask_3()
 | etask_1()
 | etask_2()
 |)
where
 use MICRO_LIB;
end;
```

This example provides a fairly simple, straightforward test. It takes the system, and spins-off three parallel tasks that run indefinitely. This is not expected to cause or encounter problems on the original system, and indeed SIGNAL detected no conflicts.

4.6.3.4 Error Detection

In the original project, the three tasks ran independently of one another, so no direct conflict is truly possible. In this next example, however, an interaction between btask3 and etask1 has been added to the system. This system tries to have btask3 accept as input a task's type, tes2, and update that type, putting the result in tes1. Likewise, etask1 attempts to take tes1, to update it, and place the result in tes2.

```
process test_driver = (?event start;!boolean fin;)
(| tes1 := btask_3(tes2)
 | tes2 := etask_1(tes1)
 | etask_2()
 | fin := (tes1 > tes2)
 |)
where
```

```
          use MICRO_LIB;
          TaskType tes1, tes2;
         end;
```

Unfortunately, this arrangement cannot function. Before either task can begin, it depends on the other to first provide their output, so the system would deadlock at runtime. SIGNAL detects this cyclic dependency, and outputs the following detected cycle:

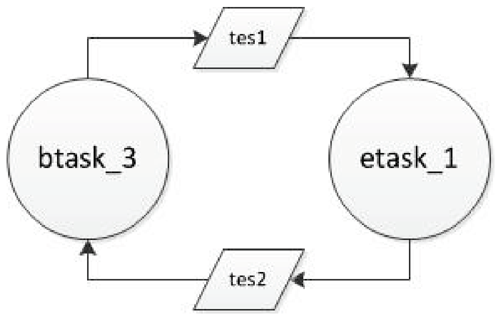

Figure 30: Overview of a Cyclic Dependency

```
| {tes1 --> tes2} when C_fin
| {tes2 --> tes1} when C_fin
```

This notation says that tes1 must precede tes2 while tes2 must precede tes1. Without buffers to store initial and generated values this is a clear conflict, and if the system were run, it would lead to system deadlock.

4.6.4 Results of the Study

These case studies have converted two actual C-based software platforms to SIGNAL, illustrating how SIGNAL can be used to build and represent practically sized systems. Additionally, the case studies were able to create scenarios for these SIGNAL implementations that either simulate flawed compositions or represent processes that cannot be safely composed. The case studies have shown that SIGNAL is able to detect such conflicts at compile time, reducing the probability of such an error going untested and causing a crash at run-time.

5 Some Paths Unexplored – Some Blind Alleys

There are some paths that were proposed to be explored, but time ran out before they could be completed. There were also some paths explored where progress was limited, so backtracking occurred. This section summarizes these areas.

5.1 Affine Clock Relation based Interface Types

If the component behavior has fixed periodicity with respect to another component, e.g. one process produces an output *x,* ten times for each time the other process is ready to consume that input, these two components are not going to compose without having an overflow of data on the channel connecting the two components. One possibility to overcome the data overflow is to design a buffer of size 10, and when the buffer gets full, the process is stopped until the other process catches up. This kind of buffer and synchronization logic synthesis can be automated if the interface types capture this numeric ratio. Standard polychronous models can express this behavior but that would require replicating the consumer process ten times, and then defining a synchronization point between the two models, which would be inefficient. If the buffer has to be synthesized, the ratio of the rate of input and rate of output must be expressed in the clock relations. Such clock relations are called affine relations.

It was planned to explore the notion of affine clock relations, and the use of such relations to check for interface compatibility, as well as to find ways to synthesize efficient buffering and synchronization logic to make them compatible. While it is believed this direction will have potential impact to efficient modeling, not enough progress was made in this area and it will be completed in a different project.

5.2 Autosar Platform Modeling Issues

For case study purposes, it was planned to extract the behavioral interfaces for an open source AUTOSAR platform model written in C++ and inject problems at the interfaces of components, and verify that the methodology could catch the injected problems. The reason for doing this was as follows: the tool to automatically extract the polychronous model from C programs was not ready for use early on, although that prototype tool is now available. So in order to demonstrate the methodology, the components were manually converted to the polychronous modeling language SIGNAL, and checked for compatibility problems. This work was described in Section 4 of this report.

The AUTOSAR platform is a large, complex system. The platform operates at a low-level, serving as an interface between applications and the underlying hardware. As a result, there is an abundance of system calls and macros that do not directly translate to SIGNAL. These calls were collected into a few supporting C library files. AUTOSAR also has several different implementations, dependent on the hardware platform being targeted for deployment. Due to limitations with the SIGNAL language, notably a lack of conditional definitions, the manual translation was completed for only one target platform: the PowerPC. Currently, that is the state of the AUTOSAR case study, as a SIGNAL-modeled PowerPC interface with supporting C libraries. The ultimate goal of this case study is to use it as a comparison of efficacy between the manually translated code and the code generated by our static single assignment based automated translation tool. The translation tool, however, is still under development and has not yet reached a point where it can translate a C program of AUTOSAR's complexity. A comparison can be made once the tool has been expanded to cover instances such as pointer-based memory access, C macros, and conditional definitions.

5.3 Ardupilot Platform Modeling Issues

For case study purposes, it was proposed to extract behavioral interfaces for Ardupilot platform models written in C++ and to inject problems at the interfaces of components, and then verify if the proposed methodology could catch those problems. This was partially performed, as described in Section 4 of this report. However, the entire Ardupilot platform was not fully modeled due to various issues.

5.3.1 Interface Variable

The Ardupilot software has a large number of static variables that are being internally modified across all of the functions. However, for a SIGNAL process to work, these static variables have to be exposed at the interface. This makes the problem of conversion very complicated since a process A that calls another process B inside it also needs to expose the static variables that are being used by process B. This results in some processes having a large number of variables in the interface. The use of modules was an approach that was tried to prevent this problem.

5.3.2 Typecasting

The Ardupilot software has various types that are not being supported by SIGNAL. This requires typecasting of variables. For example, types like uint16_t and int8_t which are used in the Ardupilot software to save space on the EEPROM are not supported in SIGNAL. Additionally, pointers are not supported in SIGNAL.

5.3.3 System Calls

There are various systems calls like delay() that are not possible to generate using SIGNAL.

5.3.4 Cyclic Function Calls

There are cases of cyclic function calls in the Ardupilot software, where Function A calls Function B and Function B calls Function A. These cases would work in C, but do not work in SIGNAL due to its Cyclic dependency.

5.3.5 Final Implementation

The above limitations make it very difficult to convert the entire Ardupilot code to SIGNAL. However, the main idea of the case study was to illustrate the issues like bugs and errors that could arise out of the implementation of the embedded system in C. The case study also tried to demonstrate how SIGNAL's clock extraction mechanism would spot the errors. Hence, various functions of Ardupilot software were converted and possible issues in the implementation in C were explored. The converted SIGNAL functions were able to identify the bugs that were not spotted during the C program analysis.

6 Conclusions & Recommendations

The primary recommendation resulting from this research is that composition of pre-designed software components without behavioral types leads to many errors including software bugs, concurrency bugs, deadlock, etc. It is difficult to subject the composed system to formal verification because most formal verification tools are not scalable to the degree required. Thus it is more efficient to check the compatibility of components by composing their interface types, and checking if the composition does not violate certain properties, such as liveness.

However, the question is: what is the best way to represent the behavior of a component's interface. The model needs to capture the behaviors of the signals/variables/events at the interface without having to completely reproduce the entire design of the component. This requires a modeling domain where model composition is not computationally explosive.

The polychronous modeling domain provides one possibility; experience from this project shows that polychrony is able to check for many properties of composition, especially liveness. However, there are other ways to express the behaviors at the interface, for example via an assume/guarantee property specification. This path was not explored in this effort. While there is reason to believe that such interfaces will be more compact, checking composition is likely to be more expensive computationally. However, this approach allows more properties of the composition to be guaranteed.

In summary, it is believed that there is a great need for continued research on the problem of software component composition and guaranteed integration results. This project was just the beginning of exploration of the space.

Appendix: Dissertations and technical reports supported by the project

The following were partially or completely supported by this project:

Bijoy A. Jose, *Formal Model Driven Software Synthesis for Embedded Systems*, PhD Dissertation, August 2011.

Jens Brandt, Mike Gemuend, Klaus Schneider, Bijoy A. Jose and Sandeep K. Shukla, *"Causality Analysis of Polychronous Programs*, FERMAT Technical Report 2011-02, 2011.

Julien Ouy, Jing Huang and Sandeep Shukla, *"Behavioral Compatibility Checking of Polychronous Components"*, FERMAT Technical Report 2011-03, 2011.

Bijoy A. Jose, Abdoulaye Gamatie, Julien Ouy and Sandeep K. Shukla, *"SMT Based False Causal loop Detection during Code Synthesis from Polychronous Specifications"*, FERMAT Technical Report 2011-04, 2011.

Jens Brandt, Mike Gemunde, Klaus Schneider, Sandeep K. Shukla and Jean-Pierre Talpin, *"Integrating System Descriptions by Clocked Guarded Actions"*, FERMAT Technical Report 2011-06, 2011.

Bijoy A. Jose, Sandeep K. Shukla, *"New Techniques for Sequential Software Synthesis from a Polychronous Data Flow Formalism"*, FERMAT Technical Report 2011-07, 2011.

Bijoy A. Jose, Abdoulaye Gamatie, Matthew Kracht and Sandeep K. Shukla, *"Improved False Causal Loop Detection in Polychronous Specification of Embedded Software"*, FERMAT Technical Report 2011-08, 2011.

Bibliography

[1] B. A. Jose and S. K. Shukla, "An Alternative Polychronous Model and Synthesis Methodology for Model-Driven Embedded Software," *Proc. of IEEE Asia and South Pacific Design Automation Conf. (ASP-DAC 2010)*, pp. 13–18, January 2010.

[2] B. A. Jose, J. Pribble, L. Stewart, and S. K. Shukla, "EmCodeSyn: A Visual Framework for Multi-Rate Data flow Specifications and Code Synthesis for Embedded Application," *12th IEEE Forum on specification and Design Languages (FDL'09)*, pp. 1–6, September 2009.

[3] B. A. Jose, J. Pribble, and S. K. Shukla, "An Actor Elimination Technique for Efficient Embedded Software Synthesis," *To appear in the Proceedings of the International Conference on Applications of Concurrency in System Design (ACSD'10)*, Portugal July 2010.

[4] B. A. Jose and S. K. Shukla, "MRICDF : A polychronous Model for Embedded Software Synthesis," Chapter in *Synthesis of embedded software - frameworks and methodologies for correctness by construction software design*, Springer, November 2010.

[5] ESPRESSO Project, IRISA, "The Polychrony Toolset," www.irisa.fr/espresso/Polychrony.

[6] A. Gamatié, *Designing Embedded Systems with the SIGNAL Programming Language: Synchronous, Reactive Specification*. Springer-Verlag New York, 2009.

[7] B. A. Jose, J. Pribble, and S. K. Shukla, "Technical Report on MRICDF models," https://filebox.vt.edu/users/bijoyaj/files/mricdfmodels.pdf, 2010, FERMAT Technical Report 2010-01.

[8] B. A. Jose, H. D. Patel, S. K. Shukla, and J.-P. Talpin. Generating Multi-Threaded code from Polychronous Specifications. In *Synchronous Languages, Applications, and Programming (SLAP'08)*, Budapest, Hungary, April 2008.

[9] B. A. Jose, S. K. Shukla, H. D. Patel, and J.-P. Talpin. On the Deterministic Multi-threaded Software Synthesis from Polychronous Specifications. In *Formal Models and Methods in Co-Design (MEMOCODE'08)*, Anaheim, California, June 2008.

[10] B. A. Jose, A. Gamatie, J. Ouy, and S. Shukla. SMT-based false causal loop detection during code synthesis from polychronous specifications. In *MEMOCODE Conference Proceedings*, July 2011.

[11] B. Jose, B. Xue, S. Shukla, and J.-P. Talpin. An analysis of the composition of synchronous systems. In *Proceedings of the 4th International Workshop on Formal Methods for GALS Design*. Elsevier ENTCS, 2009.

[12] G. Kahn. The Semantics of a Simple Language for Parallel Programming. *Proc. of Information Processing*, pages 471–475, 1974.

[13] Roberto Lublinerman, Christian Szegedy, and Stavros Tripakis. Modular code generation from synchronous block diagrams: modularity vs. code size. In *Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '09, pages 78–89, New York, NY, USA, 2009, ACM.

[14] Philip K. McKinley, Seyed Masoud Sadjadi, Eric P. Kasten, and Betty H.C. Cheng. Composing adaptive software. *Computer*, 37:56–64, 2004.

[15] Oscar Nierstrasz and Theo Dirk Meijler. Research directions in software composition. *ACM Comput. Surv.*, 27(2):262–264, 1995.

[16] Frantisek Plasil and Stanislav Visnovsky. Behavior protocols for software components. *IEEE Transactions on Software Engineering*, 28:1056–1076, 2002.

[17] H. Kopetz and G. Bauer. The time-triggered architecture. *Proceedings of the IEEE*, 91(1):112–126, Jan 2003.

[18] F. Boussinot and R. D. Simone, "The ESTEREL language," *Proc. of the IEEE*, vol. 79, no. 9, pp. 1293–1304, September 1991.

[19] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud, "The Synchronous Data-Flow Programming Language LUSTRE," *Proc. of the IEEE*, vol. 79, no. 9, pp. 1305–1320, September 1991.

[20] N. Halbwachs, "Synchronous Programming of Reactive systems," *Kluwer Academic Publishers, Netherlands*, 1993.

[21] J.-P. Talpin, P. L. Guernic, S. K. Shukla, and R. Gupta. A compositional behavioral modeling framework for embedded system design and conformance checking. *Int. J. Parallel Program.*, 33(6):613–643, 2005.

[22] Microsoft Research. What Really Happened on Mars? http://research.microsoft.com/en-us/um/people/mbj/Mars_Pathfinder/Authoritative_Account.html.

[23] ABC News. Electronic Design Flaw Linked to Runaway Toyotas http://abcnews.go.com/Blotter/toyota-recall-electronic-design-flaw-linked-toyota-runaway-acceleration-problems/story? id=9909319 .

[24] Esterel technologies. SCADE Display On-Board the Airbus A380 and A400M http://www.esterel-technologies.com/technology/success-stories/airbus-display.

[25] SMT Solvers SMT solver page at University of Iowa http://goedel.cs.uiowa.edu/smtlib/solvers.html.

[26] Bittencourt, G., Combining syntax and semantics through prime form representation. of Logic and Computation, **18**, (2008) 13–33.

[27] Coudert, O. and Madre, J. Implicit and incremental computation of primes and essential implicant primes of boolean functions. In *Proceedings of the 29th ACM/IEEE Design Automation Conference*, (1992) 36-39.

[28] de Kleer, J. An improved incremental algorithm for computing prime implicants. *Proceedings of AAAI-92*, San Jose, CA, (1992) 780–785.

[29] Fredkin, E., Trie memory, Communications of the ACM, **3**,*9* (1960), 490–499.

[30] Jackson, P. Computing prime implicants incrementally. *Proceedings of the 11th International Conference on Automated Deduction*, Saratoga Springs, NY, June, 1992. In *Lecture Notes in Artificial Intelligence*, Springer-Verlag, Vol. 607 (1992) 253-267.

[31] Jackson, P. and Pais, J., Computing prime implicants. *Proceedings of the 10th International Conference on Automated Deductions*, Kaiserslautern, Germany, July, 1990. In *Lecture Notes in Artificial Intelligence*, Springer-Verlag, Vol. 449 (1990), 543-557.

[32] Kean, A. and Tsiknis, G. An incremental method for generating prime implicants/implicates. *Journal of Symbolic Computation* **9** (1990), 185-206.

[33] Manquinho, V.M., Flores, P.F., Silva, J.P.M. and Oliveira, A.L. Prime implicant computation using satisfiability algorithms. of the International Conference on Tools with Artificial Intelligence, Newport Beach, U.S.A., November, 1997", 232–239.

[34] A. Matusiewicz, N.V. Murray and E. Rosenthal. Prime implicate tries. *Proceedings of the International Conference TABLEAUX 2009 - Analytic Tableaux and Related Methods*, Oslo, Norway, July 2009. Lecture Notes in Artificial Intelligence, Springer-Verlag. Vol. 5607, 250-264.

[35] A. Matusiewicz, N.V. Murray, and E. Rosenthal. Trie-based subsumption and improving the *pi*-trie algorithm. In *Workshop on Practical Aspects of Automated Reasoning. (Part of IJCAR 2010 within FLoC 2010), Edingurgh, UK, July 2010.*, 2010.

[36] A. Matusiewicz, N.V. Murray, and E. Rosenthal. Tri-based set operations and selective computation of prime implicates. In *Proc. International Symposium on Methodologies for Intelligent Systems - ISMIS, Warsaw, Poland, June, 2011*, 2011. Lecture Notes in Artificial Intelligence, Springer-Verlag. Vol 6804, 203-213.

[37] Morrison, D.R. PATRICIA — practical algorithm to retrieve information coded in alphanumeric. *Journal of the ACM*, **15**,4, 514–34, 1968.

[38] Ngair, T. A new algorithm for incremental prime implicate generation. *Proc of IJCAI-93*, Chambery, France, (1993).

[39] Ramesh, A., Becker, G. and Murray, N.V. CNF and DNF considered harmful for computing prime implicants/implicates. *Journal of Automated Reasoning* **18**,3 (1997), Kluwer, 337–356.

[40] Reiter, R. and de Kleer, J. Foundations of assumption-based truth maintenance systems: preliminary report. *Proceedings of the 6th National Conference on Artificial Intelligence*, Seattle, WA, (July 12-17, 1987), 183-188.

[41] Slagle, J. R., Chang, C. L. and Lee, R. C. T. A new algorithm for generating prime implicants. *IEEE transactions on Computers* **C-19**(4) (1970), 304-310.

[42] Strzemecki, T. Polynomial-time algorithm for generation of prime implicants. *Journal of Complexity* **8** (1992), 37-63.

[43] De Alfaro, L., Henzinger, T. A. "Interface theories for component-based design". *International Workshop on Embedded Software*. Lecture Notes in Computer Science v. 2211. Springer-Verlag, 2001.

[44] The yices smt solver - b. dutertre and l. de moura, http://yices.csl.sri.com/.

[45] P. Amagbégnon, L. Besnard, and P. Le Guernic. Implementation of the data-flow synchronous language signal. In *Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation*, PLDI '95, pages 163–173, New York, NY, USA, 1995, ACM.

[46] A. Benveniste, B. Caillaud, and P. Le Guernic. From synchrony to asynchrony. In J. Baeten and S. Mauw, editors, *CONCURÃ,â™99 Concurrency Theory*, volume 1664 of *Lecture Notes in Computer Science*, pages 776–776. Springer Berlin / Heidelberg, 1999.

[47] A. Benveniste, B. Caillaud, and P. Le Guernic. Compositionality in dataflow synchronous languages: specification and distributed code generation. *Inf. Comput.*, 163:125–171, November 2000.

[48] T. Gautier, P. L. Guernic, and L. Besnard. Signal: A declarative language for synchronous programming of real-time systems. In *Proc. of a conference on Functional programming languages and computer architecture*, pages 257–277, London, UK, 1987, Springer-Verlag.

[49] J. Ouy, J.-P. Talpin, L. Besnard, and P. Le Guernic. Separate compilation of polychronous specifications. *Electron. Notes Theor. Comput. Sci.*, 200:51–70, February 2008.

[50] D. Potop Butucaru, B. Caillaud, and A. Benveniste. Concurrency in Synchronous Systems. *Formal Methods in System Design*, 28:111–130, 2006.

[51] F. Remondino and N. Borlin. Polylib - a library of polyhedral functions. In *Int. Archives of Photogrammetry, Remote Sensing and Spatial Information Sciences, Vol. XXXIV, H.-G. Maas and D. Schneider (Eds)*, 2004.

[52] J.-P. Talpin and P. Guernic. An algebraic theory for behavioral modeling and protocol synthesis in system design. *Form. Methods Syst. Des.*, 28:131–151, March 2006.

[53] J.-P. Talpin, J. Ouy, L. Besnard, and P. Le Guernic. Compositional design of isochronous systems. In *Proceedings of the conference on Design, automation and test in Europe*, DATE '08, pages 928–933, New York, NY, USA, 2008, ACM.

[54] R.Tamassia. *Handbook of Graph Drawing and Visualization (Discrete Mathematics and Its Applications)*. Chapman & Hall/CRC, 2007.

List of Symbols, Abbreviations and Acronyms

3-CNF	3-conjunctive normal form
ADC	analog-to-digital converter
AFRL	Air Force Research Laboratory
APM	Ardupilot Mega
AUTOSAR	Automotive Open System Architecture
CAN	Controller area network
CFG	Control Flow Graph
CTS	C to SIGNAL
dag	directed acyclic graph
ECU	engine control unit
EmCodeSyn	Embedded Code Synthesis
GCC	GNU Compiler Collection
GNU	GNU’s Not Unix
GPS	Global Positioning System
iff	if and only if
IMU	Inertial Measurement Unit
INRIA	Institut National de Recherche en Informatique et Automatique
LIN	local interconnect network
MRICDF	Multi-Rate Instantaneous Channel Connected Data Flow Actor Model
NP-Hard	Non-deterministic polynomial time hard
OASD(R&E)	Office of the Assistant Secretary of Defense for Research and Development
OSD	Office of the Secretary of Defense

PI – Prime Implicate

PWM – Pulse Width Modulated

RC – radio control

SAT - Satisfiability

SMT – SAT Modulo Theory

SSA – Static Single Assignment

UAV – Unmanned Aerial Vehicle

USART – Universal Synchronous-Asynchronous Receiver/Transmitter

USB – Universal Serial Bus

V&V – Verification and Validation